PhD-FSTM-2023-132
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 14/11/2023 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Chánh Đức NGÔ
Born on 18 July 1989 in TP. Hồ Chí Minh, Viet Nam

AUTOMATED TESTING OF SOFTWARE UPGRADES FOR ANDROID SYSTEMS

Dissertation defence committee

Dr Lionel BRIAND, dissertation supervisor
Professor, Université du Luxembourg

Dr Fabrizio PASTORE, Vice Chairman
Professor, Université du Luxembourg

Dr José Miguel ROJAS
Professor, University of Sheffield

Dr Jacques KLEIN, Chairman
Professor, Université du Luxembourg

Dr Oliviero RIGANELLI
Professor, University of Milano Bicocca

Acknowledgement

I would like to express my heartfelt gratitude to my esteemed supervisors, Lionel BRIAND and Fabrizio PASTORE, for their unwavering guidance, invaluable insights, and relentless support throughout the entirety of this research journey. Your expertise, patience, and constructive feedback have played a pivotal role in shaping the direction and quality of this dissertation.

I am deeply indebted to my parents for their endless encouragement, sacrifices, and unwavering belief in my abilities. Your unwavering support and unconditional love have been my pillars of strength, motivating me to overcome challenges and persevere towards the completion of this endeavor.

To my beloved friends, whose encouragement, camaraderie, and understanding provided a much-needed respite during moments of stress, I extend my heartfelt appreciation. Your presence in my life has made this journey not only academically enriching but also emotionally rewarding.

Furthermore, I extend my gratitude to University of Luxembourg, particularly Interdisciplinary Centre for Security, Reliability and Trust (SnT) and Huawai Co. Ltd China, whose financial and facility supports were instrumental in the successful execution of this research.

Last but not least, I am thankful to all those who contributed in any way to this work, whether through thought-provoking discussions, suggestions, or moral support. Your collective contributions have played a crucial role in shaping the outcome of this dissertation.

In conclusion, this dissertation stands as a testament to the collaborative effort and support of many individuals and entities. Your belief in me and dedication to my success have left an indelible mark on this work, and for that, I am truly grateful.

Chanh Duc Ngo
University of Luxembourg
November 21, 2023

Abstract

Apps' pervasive role in our society motivates researchers to develop automated techniques ensuring dependability through testing. However, although App updates are frequent and software engineers would like to prioritize the testing of updated features, automated testing techniques verify entire Apps and thus waste resources. Further, most testing techniques can detect only crashing failures, necessitating visual inspection of outputs to detect functional failures, which is a costly task. Despite efforts to automatically derive oracles for functional failures, the effectiveness of existing approaches is limited. Therefore, instead of automating human tasks, it seems preferable to minimize what should be visually inspected by engineers.

To address the problems above, in this dissertation, we propose approaches to maximize testing effectiveness while containing test execution time and human effort.

First, we present *ATUA (Automated Testing of Updates for Apps)*, a model-based approach that synthesizes App models with static analysis, integrates a dynamically-refined state abstraction function, and combines complementary testing strategies, thus enabling ATUA to generate a small set of inputs that exercise only the code affected by updates. A large empirical evaluation conducted with 72 App versions belonging to nine popular Android Apps has shown that ATUA is more effective and less effort-intensive than state-of-the-art approaches when testing App updates.

Second, we present *CALM (Continuous Adaptation of Learned Models)*, an automated App testing approach that efficiently tests App updates by adapting App models learned when automatically testing previous App versions. CALM minimizes the number of App

screens to be visualized by software testers while maximizing the percentage of updated methods and instructions exercised. Our empirical evaluation shows that CALM exercises a significantly higher proportion of updated methods and instructions than baselines for the same maximum number of App screens to be visually inspected. Further, in common update scenarios, where only a small fraction of methods are updated, CALM is even quicker to outperform all competing approaches more significantly.

Finally, we minimize test oracle cost by defining strategies for selecting, for visual inspection, a subset of the App outputs. We assessed 26 strategies, relying on either code coverage or action effect, on Apps affected by functional faults confirmed by their developers. Our empirical evaluation has shown that our strategies have the potential to enable the identification of a large proportion of faults. By combining code coverage with action effect, it is possible to reduce oracle cost by about 41.2% while enabling engineers to detect all the faults exercised by test automation approaches.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Software applications for mobile devices, or Apps, play an essential role in our daily lives, from leisure to business. Therefore, App development plays an important role in our economy, with App developers paying much attention to marketing and user satisfaction. As a consequence, to improve users' experience and fulfill marketing strategies, App developers release updates frequently [1, 2, 3].

Frequent App updates increase the effort invested by developers in App testing. Indeed, every new App version should be carefully tested to avoid releasing new features that are faulty or breaking existing features. Unfortunately, App testing is expensive (i.e., around 23% - 35% of the entire project expense on average [4] and frequent App releases increase such costs even more.

Since Apps are event-driven, UI tests are preferred over unit tests, which may require complicated scaffolding in the App context; also, they enable end-to-end validation. For UI testing, developers rely on automation frameworks (e.g., Espresso for Android) to define executable UI test cases, including inputs and oracles. These test cases can be rerun to ensure the correctness of functionalities every time changes are introduced during the development.

Compared to manual testing, although this approach saves manual effort when tests are repeated, it comes with maintainability costs [5]. Indeed, in the case of updates that modify the GUI, the executable test cases manually defined by engineers may become obsolete. Further, engineers need to implement new test cases for every feature added to the updated App.

To address the problems above, abundant research on the automated testing of Apps has emerged. These approaches can be grouped according to the strategy adopted to generate test inputs, including random, evolutionary and model-based [5, 6]. Some recent approaches also adopt reinforcement learning and deep learning to guide the testing effectively [7, 8, 9, 10, 11, 12]. Unfortunately, state-of-the-art automated App testing techniques do not target updated features and show limited code coverage capabilities, thus indicating they are unlikely to exercise all the App features under test. For example, some studies report that they exercise around half of the App methods [13, 14]. Regression test selection techniques [15, 16] are unlikely to help engineers select test cases that exercise the updated features when coverage is limited. Even in the most optimistic scenario, these techniques are limited to exercising only modified functions; they do not enable test automation for features introduced by an update. Therefore, the automated testing of updated features remains an open problem.

Furthermore, to increase testing effectiveness and efficiency when updates are frequent, automated App testing approaches should leverage the knowledge, typically inferred App models, acquired when testing previous App versions. In the literature, inferred models have been reused to repair test scripts [17], execute test cases on different platforms [18, 19], and automated regression testing [20]. However, these approaches are only suitable for regression testing and do not allow effective exploration of the inferred model to generate test inputs for an updated version of the App under test (hereafter, *AUT*).

Finally, App testing, like any form of software testing, remains affected by the oracle problem [21], that is, the problem of automatically determining the correctness of a software output given an arbitrary input. The lack of techniques to automate test oracles makes the visual inspection of test outputs unavoidable. Indeed, most test automation techniques detect only crashes [22]. Oracle automation techniques address Android-related faults like data loss [23, 24], UI display issue [25], cross-device compatibility [26], setting-related bugs [27, 28, 29]. Only two tools target generic functional faults [22, 30]. Nevertheless, a recent study on functional faults affecting Android Apps reports that 98% of the failures likely require visual inspection to be detected. Among these, content-related issues account for 21%, structure-related issues for 40%, incorrect interaction for 19%, and functionality not taking effect (e.g., the end-user presses on a button, but the App unexpectedly does not react) for 12% [31]. Unfortunately, the visual inspection of App outputs is practically infeasible when automated testing tools generate a large number of test inputs, each one leading to a new output screen to be inspected. Hence, keeping the number of test inputs to a strict minimum is important to minimize human intervention. Further, selecting a subset of outputs which are representative could help reduce human effort.

All the problems above motivate our work, which has been supported by Huawei Co. LTD, a leader in mobile devices and software development, who provided constructive feedback and assessed the developed solutions.

To cost-effectively test App updates by reusing models and minimizing oracle costs, we propose (1) ATUA (Automated Testing of Updates for Apps), a test automation technique that combines static and dynamic program analysis together with an incremental testing strategy, (2) CALM (Continuous Adaptation of Learned Model), a technique that enables the reuse of inferred models acquired when testing a previous App version to drive the testing on an updated App version, and (3) a set of strategies for the selection of relevant outputs to minimize the cost of visual inspection (test oracles).

1.2 Research Contributions

Over the last decade, Android has consistently been chosen by the majority of users. A recent study shows that in July 2023, Android accounts for 70.9% of the mobile operating system market share worldwide, while iOS accounts for 28.36%[1]. Since Android is open source and provides a more flexible experiment environment for developers, it is the best candidate for test automation technique research. Indeed, about 89% of research works were conducted on Android, while only about 4% were designed to work on iOS [32]. To facilitate comparisons with related work, in this dissertation, we focus on testing of Android Apps, although the proposed approaches should be applicable to Apps developed for other operating systems (e.g., iOS, HarmonyOS). Our research contributions are described below.

[1] Statcounter - GlobalStates URL:https://gs.statcounter.com/os-market-share/mobile/worldwide

First, we introduce *ATUA (Automated Testing of Updates for Apps)*, the first approach to focus on App updates and integrating multiple test strategies to efficiently use the test budget. ATUA generates models of the AUT by combining static and dynamic program analysis. It introduces a dynamically refined state abstraction function that refines the states of the AUT during testing. To deal with the complexity of Apps, ATUA implements complementary testing strategies, including (1) coverage of the *model structure*, (2) coverage of the *App code*, (3) *random* exploration, and (4) coverage of *dependencies* among App windows.

Second, we present *CALM (Continuous Adaptation of Learned Models)*, an App testing technique that efficiently tests updated Apps by relying on models learned from previous App versions. CALM leverages ATUA to select test inputs that exercise updated methods. However, CALM improves over ATUA by combining dynamic and static program analysis to adapt and improve the model learned when testing a previous App version. The reuse of an existing model enables CALM to efficiently use the test budget to exercise updated methods rather than to determine, with random exploration, how to reach Windows already reached in previous App versions.

Finally, we propose and assess different strategies to reduce the cost of test oracles based on visual inspection. The proposed strategies rely on either code coverage or GUI screenshots, which could be acquired by any testing tool, thus enabling our findings to generalize beyond CALM. Our empirical assessment is based on functional faults confirmed in App repositories.

1.3 Dissemination

ATUA was published in ACM Transactions on Software Engineering & Methodology [33]. The work was presented at the Journal First track of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022). The ATUA toolset has been presented in the tool demo track of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022). ATUA is open source and is available online [2].

CALM has been submitted in ACM Transactions on Software Engineering & Methodology. A preprint copy of the work has been uploaded on Arvix.

[2] https://github.com/SNTSVV/ATUA

1.4 Dissertation Outline

In Chapter 2, we present background information on App design and testing, together with related work on App testing. We introduce two common types of techniques for App Testing, including test input generation techniques and App regression testing techniques. We explain why these techniques are inadequate in the context of testing Apps updates.

In Chapter 3, we present ATUA (Automated Testing of Updates for Apps), which leverages static and dynamic analysis for constructing App models to guide the testing under complementary testing strategies targeting App updates. We address three research questions: (1) Can ATUA reduce the human effort required for testing Apps, compared to state-of-the-art approaches? (2) Can ATUA effectively test Apps within practical time budgets, compared to state-of-the-art approaches? (3) Is there any difference in the functionalities that are automatically exercised across test automation approaches?

In Chapter 4, we present CALM (Continuous Adaptation of Learned Models), an App testing technique that efficiently tests updated Apps by relying on models learned with previous App versions. We conducted an empirical evaluation addressing two research questions: (1) Is CALM more effective than competing approaches in testing App updates, for a same test budget? (2) How do CALM and competing approaches fare, for different testing time budgets, with updates of different magnitudes?

In Chapter 5, we present different strategies that minimize the oracle cost by selecting the App outputs to be visually inspected. Further, we conducted an empirical evaluation addressing three research questions: (1) Do target action screenshots (i.e., screenshots recorded after triggering updates), enable the detection of functional faults through visual inspection? (2) How effective is CALM in exposing functional failures? (3) What is the most cost-effective output selection strategy?

Chapter 2

Background and Related Work

2.1 App Design and Architecture

In this dissertation, we target Android Apps since Android is the most adopted platform and is widely investigated in research [2]. However, most of the solutions proposed here should be easily tailorable to other platforms.

When an App is running, the end-user interacts with the active *Window* of the App (i.e., the Window being rendered on the screen). Windows consist of a hierarchy tree of widgets; in this dissertation, we use the term *GUITree* to refer to such hierarchy trees [34, 35, 36, 37][1]

A widget extends the class *View*. Figure 2.1 shows a portion of the hierarchy tree of the class *View*. Figure 2.2 shows a portion of the GUITree for a window of Activity Diary, one of our case study Apps (see Section 4.3).

Each widget has a set of properties associated to it. Widgets can be associated to *EventHandlers* that are invoked by the OS when specific *InputEvents* are triggered by the end-user. Typical InputEvents include click, long click, swipe, and keypress.

In Android, the application logic is typically implemented by Activity classes that are

[1] Other work uses the term GUITree to refer to the sequence of App windows encountered during testing [38]

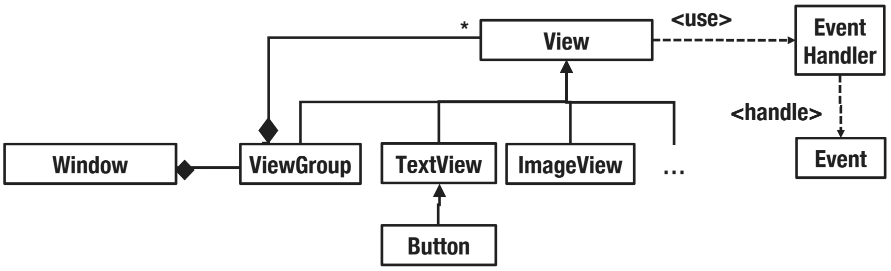
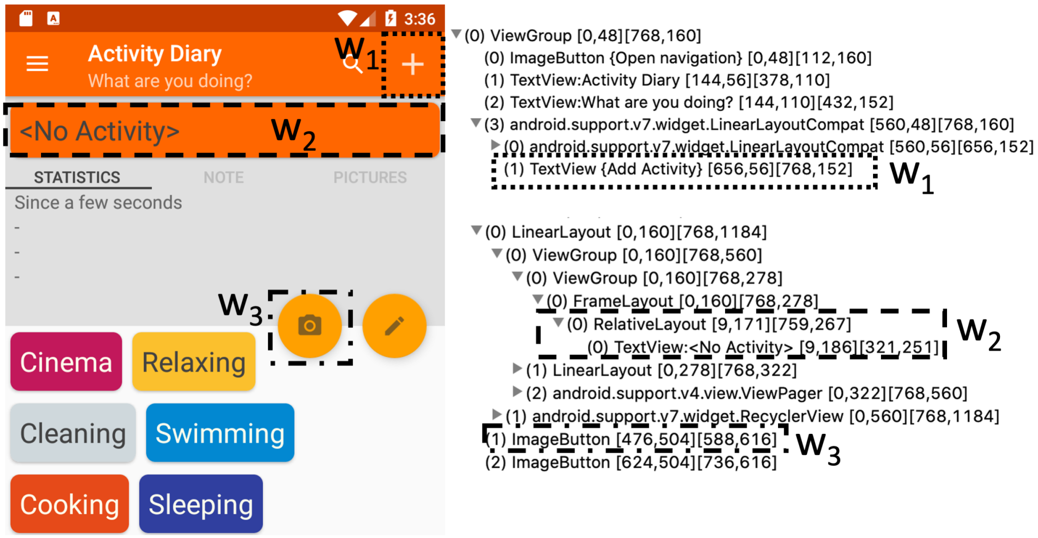


Figure 2.1: Overview of the Android Apps's GUI architectural components.



```
▼(0) ViewGroup [0,48][768,160]
    (0) ImageButton {Open navigation} [0,48][112,160]
    (1) TextView:Activity Diary [144,56][378,110]
    (2) TextView:What are you doing? [144,110][432,152]
  ▼(3) android.support.v7.widget.LinearLayoutCompat [560,48][768,160]
    ▶(0) android.support.v7.widget.LinearLayoutCompat [560,56][656,152]
     (1) TextView {Add Activity} [656,56][768,152]

  ▼(0) LinearLayout [0,160][768,1184]
    ▼(0) ViewGroup [0,160][768,560]
      ▼(0) ViewGroup [0,160][768,278]
        ▼(0) FrameLayout [0,160][768,278]
          ▼(0) RelativeLayout [9,171][759,267]
              (0) TextView:<No Activity> [9,186][321,251]
      ▶(1) LinearLayout [0,278][768,322]
      ▶(2) android.support.v4.view.ViewPager [0,322][768,560]
    ▶(1) android.support.v7.widget.RecyclerView [0,560][768,1184]
    (1) ImageButton [476,504][588,616]
    (2) ImageButton [624,504][736,616]
```

Figure 2.2: Example of a GUITree. We use different dotted boxes to match widgets in the tree to the pixels in the screen.

instantiated by the framework and act as controllers of the Model-View-Controller design pattern [39]. Inter-process communication, instead, is managed by the *Intent* resolution mechanism [40]. More precisely, in Android, a system event (e.g., indicating a battery being low) or a message exchanged between apps (e.g, a URL sent by the browser to a music player App) is referred to as an *Intent*. To handle Intents, an App declares in its XML configuration file, called App Manifest [41], an Activity that the OS will instantiate and execute when a specific Intent type is received by the App.

2.2 App GUI Testing Automation

2.2.1 Overview

System-level testing of an App through its GUI (i.e., GUI testing) is performed through sequences of *test inputs* that can be either Events or Intents. Functional GUI testing aims at exercising (i.e., render active) all declared Windows, trigger all event handlers, and cover all the code of the AUT. App testing automation aims to generate input sequences to achieve these objectives at the lowest cost possible.

App testing tools differ with respect to their input selection strategy [5, 32] and mostly rely on random [42], model-based [43, 36, 37, 44], search-based [45], deep learning [9, 11, 10] and reinforcement learning [7, 8, 12, 46] strategies.

2.2.2 App Model as a Finite State Machine

Although App testing solutions may integrate different state-of-the-art techniques, they, in general, rely on a finite state machine (FSM). A finite state machine can be formally described as a tuple $(S, A, T, \mathcal{L})$ [37], where

- S is a set of states.
- A is a set of actions.
- T is a set of state transitions. Each transition has a source state $s_{src} \in S$ and a target state $s_{tar} \in S$. It is triggered by an action $\alpha \in A$.
- $\mathcal{L}$ is an abstraction function, which might be used to: (1) assign a Window to a state and (2) match an InputEvent or an Intent to an action.

2.2.3 Test Generation Techniques

The most common test generation techniques are random, model-based, and evolutionary [32]. Representative approaches of these three categories used in empirical evaluations are Monkey, Stoat [36], and Sapienz [45], respectively. Other recent and effective approaches either rely on Q-Learning [47, 48], Deep Learning [9, 11, 10], execution traces [49], or state cloning [50].

- Monkey [42] is a program that runs on the Android emulator and generates pseudo-random streams of events. It is used as baseline for App testing approaches and surprisingly fares better in many benchmarks [14]. The reason is that the time saved by not processing the App GUITree can be used to further explore the App state space.
- Stoat [36] performs stochastic model-based testing. It relies on dynamic analysis based on a weighted UI exploration strategy to derive a stochastic finite state machine

(SFSM) of the App's GUI interactions. Stoat relies on the SFSM to generate test suites using an objective function that aims to maximize code coverage, model coverage, and test diversity. The test generation process relies on Gibbs sampling [51], an instance of Markov Chain Monte Carlo sampling, to iteratively mutate and refine the SFSM, based on the fitness of the generated test suite.

- **GATOR [52]** is a static analysis tool for constructing Window Transition Graphs (WTG) for Apps. It further relies on the WTG as App model to generate test input sequences as test scripts (i.e., Robotium [53] test scripts) by traversing the WTG. However, due to the limitation of the static analysis (i.e., the incompleteness of the constructed WTG), the generated test suites are likely infeasible to execute, thus requiring manual effort to fix the scripts. Further, such static-analysis-based generation approaches are usually capable of generating only short test input sequences due to the explosion of the search space, which is limited to testing scenarios requiring complex input sequences.

- **Sapienz [45]** is an evolutionary approach that utilizes Pareto multi-objective search to automatically explore and optimize test sequences, minimizing length, while simultaneously maximizing coverage and fault detection. Sapienz combines random fuzzing, systematic exploration and search-based exploration.

- **APE [37]** employs an adaptable state abstraction function to eradicate the state explosion issue (due to a selection of over-fine-grained state abstraction function, which captures every information from the GUITree) but at the same time to encourage app exploration. In APE, each window is modeled with sets of *attribute paths* that univocally identify the widgets of the window. In APE, a single $\mathcal{L}$ is defined for the whole app, and $\mathcal{L}$ is implemented by means of a decision tree (DT), which enables APE to not rely on a predefined set of reducers, each of which is a function allowing extract a piece of specific information acquired from the GUITree.

- **Q-testing [47]** relies on Q-learning (a model-free reinforcement learning algorithm [54]) to trigger events that enable the exploration of features that are not yet tested. In Q-testing, an event is rewarded if it leads to an App state having a layout not visited before. A Siamese neural network [55] is used to compare states. An event is likely to be selected based on its value in a Q-Table which stores scores of every event, or selected randomly according to a predefined probability.

- **Humanoid [56]** relies on a deep neural network model to prioritize unexplored actions in a human-like way. Humanoid trains the DNN model with multiple Residual LSTM [57] on a crowd-sourced dataset consisting of interaction flows from more than 10,000 apps. An action is likely to be selected if its probability of selection in the current context (i.e., the three latest transitions) is highest among other actions.

- **MuBot [10]** is inspired by Humanoid, but it differs from Humanoid in the such that it employs multi-model deep neural networks to provide more selection possibility of actions in the current context, then to help the testing enhance the effectiveness of GUI state abstraction and GUI action generation so that the testing would not fall into the infinite-loop issue, particularly in industrial apps whose GUI design are sophisticated.

- **DeepGUI [11]** trains a DNN model to improve Monkey's effectiveness by focusing on areas on the App screen that could induce changes in the GUI when they receive inputs (e.g., touch). The UNet neural network architecture is adopted to generate a multi-channel heatmap of the current screen. Each channel corresponds to an action type (e.g, touch, swipe). DeepGUI then relies on the heatmap to generate supported inputs (i.e., touch or gesture) for Monkey.

- **Combodroid [49]** works by combining test input sequences derived from execution traces. It can work with either traces collected by automated testing tools (e.g., APE) or traces collected when end-users exercise the AUT. In the first case, results show that Combodroid achieves higher code coverage than APE when using more than six hours of test budget, which is not feasible in continuous integration contexts where test cases are always executed after code commits.

- **TimeMachine [50]** implements a metaheuristic approach that relies on a pool of App states. New App states are reached by triggering random events. An App state is added to the pool only if it is reached after exercising code not covered yet; App states are captured by cloning the state of the virtual machine running the AUT. When lack of progress is detected, TimeMachine resets the execution from the state with the highest fitness, which is computed by balancing the number of times the state has been visited and the number of interesting states generated from it. Similar to Combodroid, TimeMachine overcomes Monkeys, Stoat, and Sapienz when executed for more than five hours; however, as discussed above, this setting makes it inapplicable in some continuous integration contexts. Most importantly, none of the existing App testing approaches prioritize the testing of App updates.

Independent empirical evaluations performed by Choudhary et al. [13] and Wang et al. [14] have reported that combinations of different testing strategies could improve the result. Further, both studies show that the method and instruction coverage achieved by all test automation approaches are relatively low, that is below 50%. Choudhary et al. [13] note that model-based approaches complement random approaches regarding fault detection, while for code coverage, random approaches fare better. Wang et al. [14] confirm these results. They report that random and evolutionary approaches are complementary regarding method coverage, while both evolutionary and model-based approaches complement random approaches in terms of fault detection. However, the validity of these findings has been weakened by recent advances in model-based approaches. Indeed, more recent results show that model-based approaches that either integrate advanced exploration strategies (i.e., biased random in DM2) or adaptable state abstraction functions (i.e., APE [37]) fare better than random approaches or state-of-the-art model-based approaches. APE, for example, is a recent technique that has been reported to perform better than Monkey, Sapienz, and Stoat.

2.3 App Regression Testing Techniques

Regression testing techniques for Apps concern the selection of events that may trigger modified code [16], the selection of regression test cases [15], and the repair of existing test suites [58].

- **QADroid** is a static analysis toolset [16] that identifies the events (i.e., the Inputs of WTG) that may trigger the execution of modified methods. It relies on FlowDroid [59]. To select the Inputs that may trigger modified methods, QADroid performs a forward traversal of the App control flow graph obtained with Soot [60] (i.e., it starts the traversal from event handler methods) and selects all the inputs that reach modified methods. QADroid has never been applied to automated testing and cannot identify the concrete input values to be used with certain Inputs (e.g., the data to write into a TextArea).

- **Redroid [61, 62]** selects a regression test suite for an updated version of an App from an existing test suite. It relies on state-of-the-art static analysis procedures [63] to perform change impact analysis and uses code coverage information to select any test case that cover modified blocks of code. Redroid does not support the generation of a minimal test suite.
- **DetReduce [15]** creates a small regression test suite for an App from a test suite generated by a model-based test automation tool. It identifies and removes redundant method call traces and subtraces within the test suite and redundant loops within a test case. Redundancy is identified based on a state abstraction function that considers actionable widgets and their visible attribute values. Widgets are identified by their path in the GUITree.

Since executable test suites are often unavailable and state-of-the-art App testing tools can cover only a narrow set of modified methods, the applicability of Redroid and DetReduce remains limited.

Another solution for the generation of reduced test suites is the one proposed by Jabbarvand et al. [64] in the context of energy-aware test-suite minimization. In their work, they stop and restart the execution of Monkey every 500 events; a sequence of 500 events is a test case. When the test budget is exhausted, a greedy algorithm identifies the minimal set of test cases that maximize the coverage of energy-greedy parts of the App. Such solutions might be adapted to work with the state-of-the-art test generation approaches (introduced in 2.2.3); for example, by restarting test generation after a predefined number of events and then by relying on the greedy algorithm to select the minimal set of test cases (e.g., up to 2000 inputs) that maximizes the code coverage. However, though such an approach might reduce the size of the generated test suites, it might not be feasible to identify a configuration that maximizes the benefits for a range of Apps. Indeed, for some Apps, 500 events might not be sufficient to reach an App state that enables exercising updated methods, especially for random approaches (e.g., if updated methods require long, specific sequences of events to be exercised), while a much larger number of events would greatly reduce the benefits of test suite reduction

Automated repair techniques for the GUI test scripts of mobile Apps are still preliminary [58, 65, 66]; to repair test scripts they include strategies ranging from static program analysis [58], to model-based [67] and computer vision techniques [65, 66]. Existing approaches either leave between 5% [66] and 8% [58] of the test scripts to be manually repaired or do not preserve all the test actions (i.e., the test semantic [67]). Though these results show that automated GUI script repair techniques might be adopted to support the oracle automation approach we suggested for the identification of regression failures, manual intervention would still be required, as indicated in Section 3.4.6.

2.4 Incremental Testing Approaches

Campos et al. [68] have been the first to propose a technique to incrementally test the units in a software project, leading to overall higher code coverage while reducing the time spent on test generation. They apply an evolutionary approach (i.e., EvoSuite [69]) and optimize test generation by providing more test budget to the modified portions of the code and reuse already generated test cases for seeding (i.e., they re-run all the test cases that compile). However, the approach does not target the GUI testing of Apps but the unit testing of Java libraries. In our context, the reuse of existing test cases is complicated by the presence of a GUI, which is likely updated across versions and may break existing test sequences.

2.5 Test Oracle Automation

Most state-of-the-art test generation approaches for App testing [5, 70] do not address the *oracle problem* [21].

Xiong et al. [31] have recently conducted an empirical study of 399 functional faults in Android Apps. Their results highlight that most of the functional faults require visual inspection to be detected. Indeed, they report that only 30% of the faults lead to crashes. Of the remaining, only 3% are related to energy consumption, the rest is probably detectable only through visual inspection; indeed, content-related issues account for 21%, structure-related issues for 40%, incorrect interaction for 19%, functionality not taking effect for 12%, and unresponsive UI element for 5%.

Recently, some studies have proposed approaches for generating tests with automated oracle. They mainly rely on one of four testing techniques [31]: differential testing, metamorphic testing, deep learning, or implicit knowledge.

- **Differential testing** [71] detects anomalies of App features by comparing test outputs between a test subject (e.g., an updated version) and a reference subject (e.g., a base version) over the same input. DiffDroid [26], ITDroid [29], and FILO [72] adopt this technique. DiffDroid identifies cross-platform inconsistency issues of an App by generating test cases to explore the App's UI and construct UI model of the App on the reference device, then re-executing the test cases on test devices to check the consistencies of UI models obtained with the reference UI model. ITDroid focuses on detecting layout changes in the context of Apps' internationalization by following a similar process, except that it relies on *layout graph* of default language as a baseline to detect changes in other languages. FILO collects and analyzes execution traces including data exchanged between Android APIs and App callbacks when executing the App on two Android versions to detect backward incompatibility issues introduced by the upgrade of the Android framework.

- **Metamorphic testing** [73] focuses on verifying the correctness of a program by applying transformations to its inputs and comparing the outputs with expected behaviors, relying on the principle that certain transformations should yield predictable, consistent results. Approaches like SetDroid [27] and Genie [22] define metamorphic relations and generate mutant tests whose results are supposed to satisfy the metamorphic relations. SetDroid addresses the inconsistent behaviors of Apps when there are changes in Android Settings (i.e., Setting-related faults), while Genie focuses on apps' generic functional faults that violate "independent-view property" metamorphic relation.

- **Deep learning** is adopted to detect common UI display issues in Owl Eyes [25]. Owl Eyes relies on a CNN (Convolutional neural network) to classify whether an App screen has UI display issues or not. Similarly, Glib [74] adopts a CNN to detect glitches in graphically rich applications (i.e., Games).

- Finally, some techniques rely on **implicit knowledge**. iFixDataloss [23] and DLD [24] assume that a GUI state should be preserved after a sequence of neutral actions (e.g., after a double rotation). Otherwise, a data loss fault is present. Odin [30] targets functional bugs by relying on the viewpoint that considers "bugs as deviant behavior". When there is a large number of observations, an action effect that rarely happens is likely the presence of a fault.

Further, Xiong et al. also report that feature agnostic oracles (e.g., looking for overlapping UI elements, data loss, and App freezing) discover 30% of non-crashing functional failures; however, in their experiments, existing tools automating feature agnostic oracles (i.e., Genie [22], Odin [30], IFixDataLoss [23], ITDroid [29], and SetDroid [27]) could, overall, detect only 6% of such faults. In other words, 84% of non-crashing failures can be detected only through visual inspection, which further motivates our work.

Finally, Xiong et al. propose RegDroid, which implements a form of differential testing to detect regression faults (i.e., it executes a random sequence of actions on two App versions and verifies if the output screens include a same randomly selected interactable widget); unfortunately, their results show a false positive rate above 60%.

The automated detection of functional regression faults based on the identification of unexpected changes to outputs across different software versions is not new in software testing [75, 76, 77, 20]. However, since regression faults are a subset of all the possible functional faults in an App, engineers still need to inspect App outputs for non-regression faults; therefore, for engineers, it may likely be more effective to simply inspect all the outputs generated by an App.

One last solution to visually inspect App outputs while containing costs, consists of relying on crowdsourcing, a popular solution adopted by industry to reduce the costs of manual GUI testing for Apps [78, 79]. Crowdsourcing consists of delegating tasks that are only accomplished by humans and often do not require expertise in a specific domain to the public (e.g., in the country or worldwide) through online platforms. Related work has shown that it is feasible for crowd workers to identify errors after visualizing the inputs and outputs of the functions under test [80]; in the App context, crowd oracles may lower testing costs while test coverage is addressed by test automation.

Chapter 3

ATUA: Automated Testing of Updates for Apps

3.1 Introduction

The business-critical role played by software applications for mobile devices (Apps) in our society [81] has led to the development of dedicated techniques for their automated testing [5]. Since most of the code in an App concerns the handling of input values and events, test automation approaches automatically generate sequences of events and input values (hereafter, input sequences) that simulate the use of the AUT in its deployed environment. These approaches mainly differ with respect to the strategy used to create input sequences, such as random, evolutionary, and model-based approaches relying either on static or dynamic information [5].

Unfortunately, state-of-the-art automated App testing techniques show limited code coverage capabilities, thus indicating they are unlikely to exercise all the features of the AUT. For example, they typically exercise about half of the methods implemented by commercial apps [14]. As a result, all methods and instructions that are not automatically tested should be exercised by manually implemented test cases, an expensive task that may delay the

App release. Also, though existing techniques show a degree of complementarity [14], state-of-the-art approaches do not attempt to integrate them to achieve better results.

Existing testing approaches do not account for the high release frequency of a typical App's lifecycle, which is usually driven by marketing strategies aiming at increasing visibility [3, 1, 2]. As a result, existing work does not include effective means of prioritizing the testing of modified or newly introduced features and is thus not addressing one of the major needs of App developers. However, this is an important requirement for any testing strategy as exercising all the features of an App in each release is enormously wasteful. Existing work on testing App upgrades is limited to the selection of subsets of events that may trigger modified code [16] or the selection of regression test cases [15]. This is, however, not adequate when, to start with, available test cases do not exercise all the new and modified features of the software.

Finally, the current body of work does not address the *oracle problem* [21, 5, 70]. More precisely, testing techniques cannot discover functional failures beyond crashes and the manual verification of the App outputs is difficult due to the large number of inputs they exercise [15]. However, in the context of frequent App updates, with a test input generation strategy that effectively exercises updated features, it is conceivable to address the oracle problem by relying on dedicated strategies to minimize test inputs. Failures affecting unchanged features (i.e., regressions) can be automatically detected by comparing the output of different App versions for a same input [75, 76, 77, 20], whereas the output of new and modified features can instead be verified, at reduced costs, by relying on internal or external crowdsourcing [80]. Nevertheless, such solutions are only practical if the number of test inputs is kept down to a reasonable number.

Keeping the number of test inputs to the strict minimum is important to minimize human intervention since it may be required when executing the same inputs on different software versions, e.g., to adapt input sequences to changes in the GUI [67, 65]. Further, screenshots of the results must be visualized after every input. Unfortunately, state-of-the-art App testing approaches generate large test suites, while test suite reduction approaches require to perform runtime monitoring of the App, which slows down execution and diminishes test automation effectiveness [15].

In summary, to address the limitations above, we aim to achieve the following two objectives: (O1) maximize the number of updated methods and their instructions that are automatically exercised within practical test execution time, and (O2) generate a significantly reduced set of inputs, compared to state-of-art approaches, thus decreasing human effort.

To achieve the two objectives above, it is necessary to integrate multiple analysis strategies. Objective O1 can be effectively achieved by means of static analysis, to determine updated features (e.g., through the identification of updated methods [16]) and identify the inputs that may trigger a specific feature (e.g., the input that leads to a particular Window) [82]. Unfortunately, static analysis alone may not enable the effective testing of Apps; indeed, they typically rely on APIs dedicated to input handling that are hardly processed by static analysis tools, as discussed in related work [70]. Random exploration is thus required to discover, at runtime, inputs that may trigger a potentially large subset of modified methods. Unfortunately, random exploration might be particularly inefficient and conflict with objective O2 (e.g., it may require thousands of inputs to exercise features that depend on specific App states). For this reason, it is necessary to determine which inputs bring the App into distinct program states by relying on dynamically-refined state abstraction functions [37] and by identifying dependencies among App features (e.g., to determine that an option in the settings page enables a specific feature).

We present *ATUA*[1] *(Automated Testing of Updates for Apps)*, the first approach that integrates multiple test strategies to efficiently use the test budget and achieve the two above-mentioned objectives. The rationale followed by ATUA is that Apps can be cost-effectively tested by combining static and dynamic program analysis to select the inputs that exercise updated methods, our test targets. Also, given the complexity of Apps, testing should be performed incrementally, by focusing first on objectives that are easier to achieve. For this reason, ATUA works in three phases: (1) it exercises all the features that may trigger modified methods (e.g., submitting a registration form that is processed by an updated method), (2) to maximize coverage in the presence of data-dependencies, it exercises updated features with diverse input values (e.g., a diverse set of values in a form), (3) to maximize coverage in the presence of state-dependencies, it exercises related features (e.g., submit a registration form after changing language settings). ATUA implements a model-based approach that integrates a *dynamically-refined state abstraction function* and complementary testing strategies, including (1) coverage of the *model structure*, (2) coverage of the *App*

[1] Atua is also the name of spirits in Polynesia, https://maoridictionary.co.nz/word/494

code, (3) *random* exploration, and (4) coverage of *dependencies* among App windows.

ATUA generates models of the AUT by combining static and dynamic program analysis. It extends static program analysis approaches [82] to automatically generate extended window transition graphs (EWTG), i.e., finite state machines that capture which inputs trigger window transitions and updated methods. Also, it introduces a state abstraction function that refines the states of the EWTGs to capture differences in the user interface that are not detected by means of static analysis (e.g., the presence of dynamically disabled buttons). The state abstraction function is automatically refined to eliminate or, when not possible, reduce non-determinism while minimizing the number of abstract states.

To automatically exercise Apps, ATUA relies on the generated EWTGs to identify the sequences of inputs that trigger the execution of updated methods. When there are discrepancies between the EWTGs and the observed behavior, random exploration is used to refine the former. Code coverage is used to identify the methods that require additional testing effort. Finally, using information retrieval techniques [83], ATUA identifies dependencies between App windows that may prevent the execution of certain methods.

We assume that, for every software version, engineers are interested in testing the updated methods only. However, the general principles behind ATUA can easily be adapted for other ways of characterizing change, e.g., based on impact analysis [84]. Indeed, other criteria for selecting target methods are straightforward to integrate into ATUA.

An empirical evaluation conducted with nine popular, commercial Apps shows that, compared to state-of-the-art approaches (i.e., DM2 [43], APE [37], and Monkey [42]), ATUA leads to reduced test costs. Indeed, it generates less than 70%, 4%, and 2% of the inputs generated by DM2, APE, and Monkey, respectively. By automatically exercising, on average, 2.6 instructions belonging to updated methods for every generated test inputs, ATUA is the most cost-effective approach. Further, on average, ATUA, for a same test execution budget (e.g., 1 hour test execution time), improves the method and instruction coverage achieved by the second best, state-of-the-art approach by at least 10%. The structure of this chapter is as follows. Section 3.2 provides the technical details of the proposed approach. Section 3.3 describes the ATUA toolset. Section 3.4 reports on the results of our empirical evaluation. Section 3.5 concludes the chapter.

3.2 Proposed Approach: ATUA

Within an updated App, we can distinguish among existing features (i.e, features present in previous versions of the AUT) and new features (i.e., features introduced in the AUT). Existing features can be unchanged (i.e, their functional requirements did not change), modified (i.e, their functional requirements did change), or repaired (i.e., modified because their implementation did not match its functional requirements).

In our work, we aim to automatically exercise *updated features*, including new, modified, and repaired features. More precisely, we focus on features that are implemented either by introducing new methods or by modifying existing methods [2]. In this dissertation, we use the term *updated methods* to refer to both new and modified methods, which are our test targets.

The testing activity performed by ATUA is driven by an App model with the objective of exercising a set of test targets (i.e., updated methods). The App model is initially created by static program analysis procedures and then refined during testing.

The App model metamodel is shown in Figure 3.2 and described in Section 3.2.1. It consists of three parts: (1) an Extended Window Transition Graph (hereafter, EWTG), (2) a Dynamic State Transition Graph (hereafter, DSTG), and (3) a GUI State Transition Graph (hereafter, GSTG). The three graphs are FSMs capturing how input values trigger changes in the state of the AUT. The *EWTG* models the sequences of windows being visualized after specific inputs (Events or Intents). For every input, the EWTG keeps trace of the name of the handlers associated to the input and the list of test targets that may be invoked during the execution of the input handler. The *GSTG* is a fine-grained model that captures every visual change in the GUI (e.g., the color of a button) that might be triggered by an action on the GUI. An action is an instance of an input (e.g., click on a specific Button widget). Finally, the *DSTG* models the abstract states of the visualized Windows and the state transitions triggered by events. Abstract states are identified by a state abstraction function to eliminate possible non-determinism. The DSTG plays a critical role to optimize the test budget and identify a reduced set of input events; indeed, it helps determine a correct and reduced sequence of events necessary to reach a specific Window from another one.

Figure 3.1 provides an overview of the process implemented by ATUA to test App

[2] Based on related work, 81% of the updates concern Java files, while only the remaining 19% concerns manifest files (e.g. permissions) or layout declarations in XML files [16].

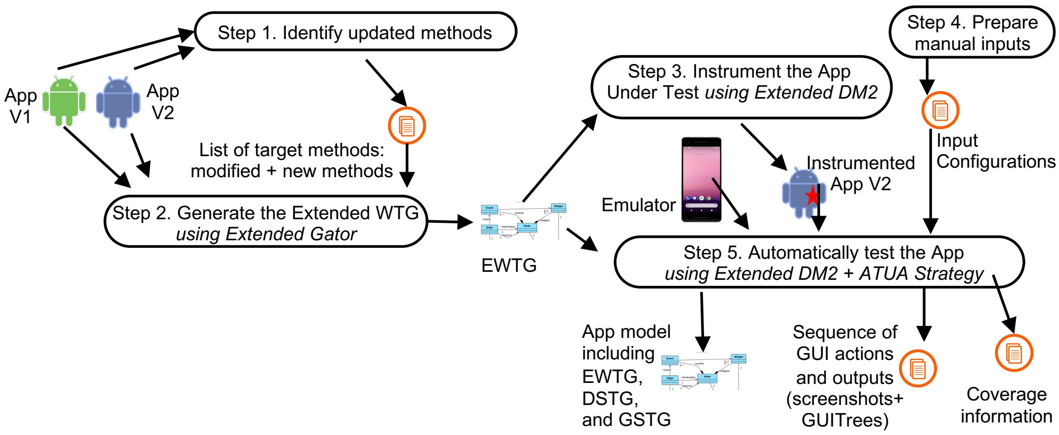

Step 1. Identify updated methods
App V1
App V2
List of target methods: modified + new methods
Step 2. Generate the Extended WTG using Extended Gator
EWTG
Step 3. Instrument the App Under Test using Extended DM2
Emulator
Instrumented App V2
Step 4. Prepare manual inputs
Input Configurations
Step 5. Automatically test the App using Extended DM2 + ATUA Strategy
App model including EWTG, DSTG, and GSTG
Sequence of GUI actions and outputs (screenshots+ GUITrees)
Coverage information

Figure 3.1: Overview of the ATUA process to test App updates.

updates. In Step 1, ATUA compares the previous (App V1 in Figure 3.1) and the updated (App V2) version of the AUT to identify the updated methods. In Step 2, ATUA relies on Soot [85], a static analysis framework, and an extended version of GATOR [52], a static analysis tool for constructing App's Window Transition Graph (see Section 3.2.5), to generate the EWTG. In Step 3, ATUA relies on DM2 to generate a version of the App that is instrumented to trace code coverage. In Step 4, engineers manually specify test inputs that are unlikely to be generated automatically (e.g., the login credentials for Apps that require a user to be registered on a remote platform). In Step 5, ATUA exercises the AUT by relying on an extended version of DM2 that integrates the ATUA test algorithm. During testing, ATUA refines the App model and relies on it to identify the actions to perform on the GUI. For example, ATUA uses the App model to identify the action that, in the current window, may lead to the execution of a test target.

The main output of Step 5 is the sequence of GUI actions performed during testing and the outputs (i.e., the screenshot of the active Window and the corresponding GUITree) generated by the AUT after every action. This sequence is used by engineers to verify if the behavior of the App is as expected (test oracle). As mentioned earlier, to verify App results, engineers can rely on two complementary state-of-the-art approaches, not addressed by ATUA, that respectively target regression failures in unchanged features and failures in newly implemented, repaired, and modified features. To discover regression failures, engineers can replicate, on a previous App version, the test input sequences generated for the updated App and automatically compare the generated outputs. Differences in the outputs generated by the two versions should indicate the presence of a regression fault. To discover failures in new and repaired features, engineers can visualize the GUITrees or the screenshots of the active Window rendered after each Action. The visual inspection of the App outputs enables an engineer to determine the presence of functional failures, based on expected behavior, whether specifications are implicit or documented. For example, the engineer shall determine if the Window rendered after each Action includes the expected content and is well positioned. Also, the engineer shall inspect GUITrees to determine if the widgets within a Window have the expected properties. In Section 4.3, we discuss to what extent ATUA reduces the cost associated to the manual activities entailed by the test oracle strategies above, with respect to other state-of-the-art test automation solutions.

In addition, ATUA provides, as output of Step 5, an App model including the EWTG, the DSTG, and the GSTG. The App model is generated and continuously refined during testing. Further, it reports coverage information, i.e., the sets of updated methods and instructions belonging to updated methods that have been exercised during testing.

In the following, we provide additional details about the App model metamodel (Section 3.2.1) and describe Steps 1 to 5 (Sections 3.2.2 to 3.2.5), except for Step 3 which is already automated by DM2.

3.2.1 App Model Metamodel

Figure 3.2 shows the ATUA metamodel as a UML class diagram. Figure 3.3 shows an example App model built when testing Activity Diary.

The EWTG is consistent with the WTG generated by GATOR. Each WindowTransition is triggered by an *Input*, either an *InputEvent* or an *Intent*. An InputEvent is associated to the Widget that declares its EventHandler. If the EventHandler is not declared by a Widget (e.g., for the event *PressHome*), the InputEvent is not associated to any target Widget. Each Widget belongs to one Window.

In addition to the concepts captured by the WTG, the EWTG generated by ATUA also captures the list of modified methods that can be triggered by the Input (i.e., the attribute *targetMethods* of class *Input*), which are used to drive testing. A WindowTransition triggered by Inputs with associated *targetMethods* is a *target Transition*. Similarly, a Window that is the source for at least one target Transition is a *target Window*. *TargetMethods* are identified by our GATOR extension (see Section 3.2.3). The EWTG also captures the dialogs and menus that can be opened by an Activity (e.g., association *triggeredDialogs*). Finally, it also models the HiddenHandlers of a Window, which are introduced in Section 3.2.3.

The GSTG captures the same information provided by the models generated by DM2. The state of a Window is captured by its GUITree, which is a composition of Widgets. For each Widget, we record the values associated to its properties and derive the Widget hash (to associate an ID to the current state of the Widget) according to the DM2 strategy. The hash of the GUITree is then derived from the hash of its Widgets. In addition, the GSTG also captures the name of the Activity running when the GUITree is visualized (see attribute *activityName*), which we derive, at runtime, from logcat [86]. The transition between GUITrees is triggered by an Action, which can be handled either by the Widget (*WidgetAction*) or by the visualized Window (*WindowAction*). The enumerations *WidgetActionType* and *WindowActionType* list the type of actions that can be performed to trigger GUITreeTransitions. Actions might have additional information (i.e., actionData) associated to them; for example, the text provided to the AUT by a TextInput action or the start and end coordinates of a Swipe action.

The DSTG provides abstract states that group together GUITrees in which a same Action triggers a same App behaviour (e.g., leads to a same abstract state). The DSTG enables ATUA to efficiently test the AUT by determining the shortest sequence of Actions that reaches a target Window. Also, abstract states capture the conditions under which a specific Action can trigger a modified method, for a certain Window. Abstract states are thus a mean to minimize the number of Actions generated by the test automation approach.

Each AbstractState consists of a number of AttributeValuationMaps, each one abstracting the state of the widgets belonging to the GUITree that have the same set of attribute valuations. An *AttributeValuationMap* is a map of pairs ⟨attribute,value⟩. The enumeration *Attribute* in Figure 3.2 provides the list of attributes appearing in the AttributeValuationMap. The AttributeValuationMap has a cardinality attribute, which indicates how many widgets have the same attribute valuations. For example, in Figure 3.3, the AbstractState *as6* includes many LinearLayout widgets (*LL* in the Figure, cardinality *MANY*), one for each item in the displayed list.

Since the DSTG is used to drive testing, i.e., to select the Actions to be triggered at runtime, the AbstractState captures only the attributes of widgets that are *interactive*. A widget is *interactive* when it is enabled, visible, and is an instance of a class that can be the

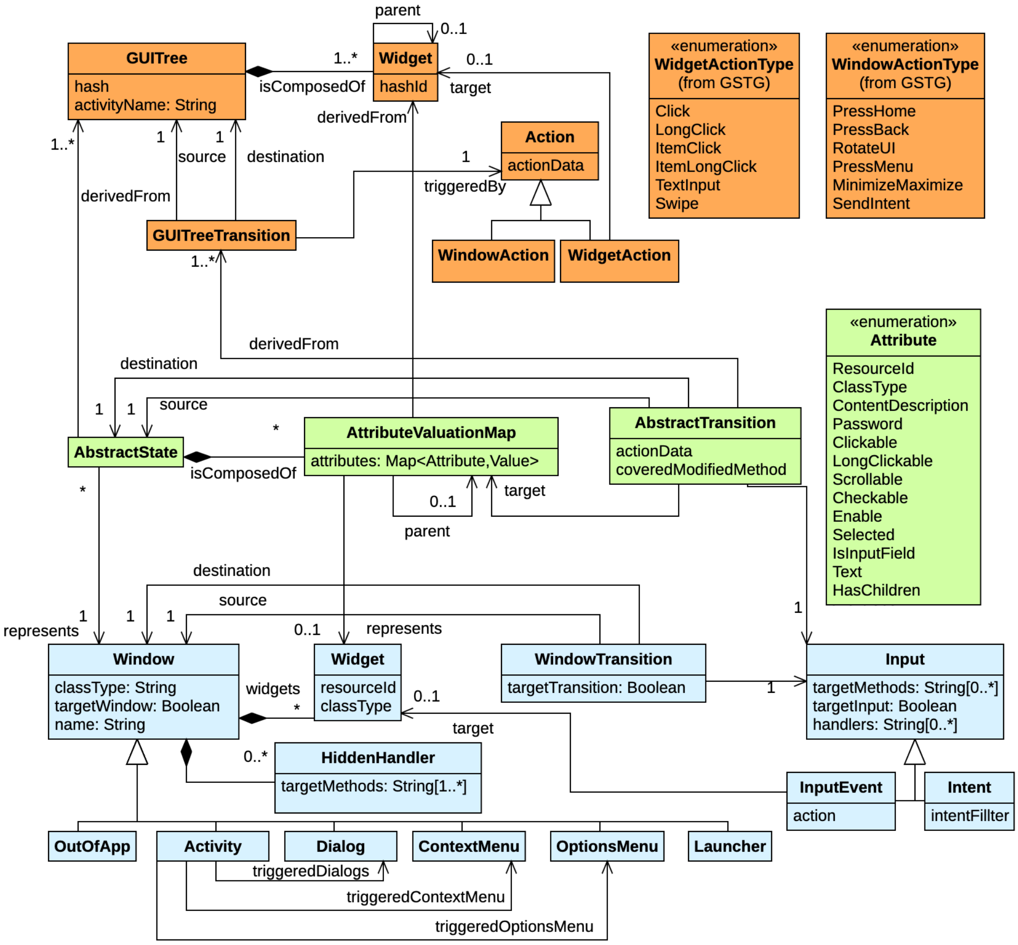
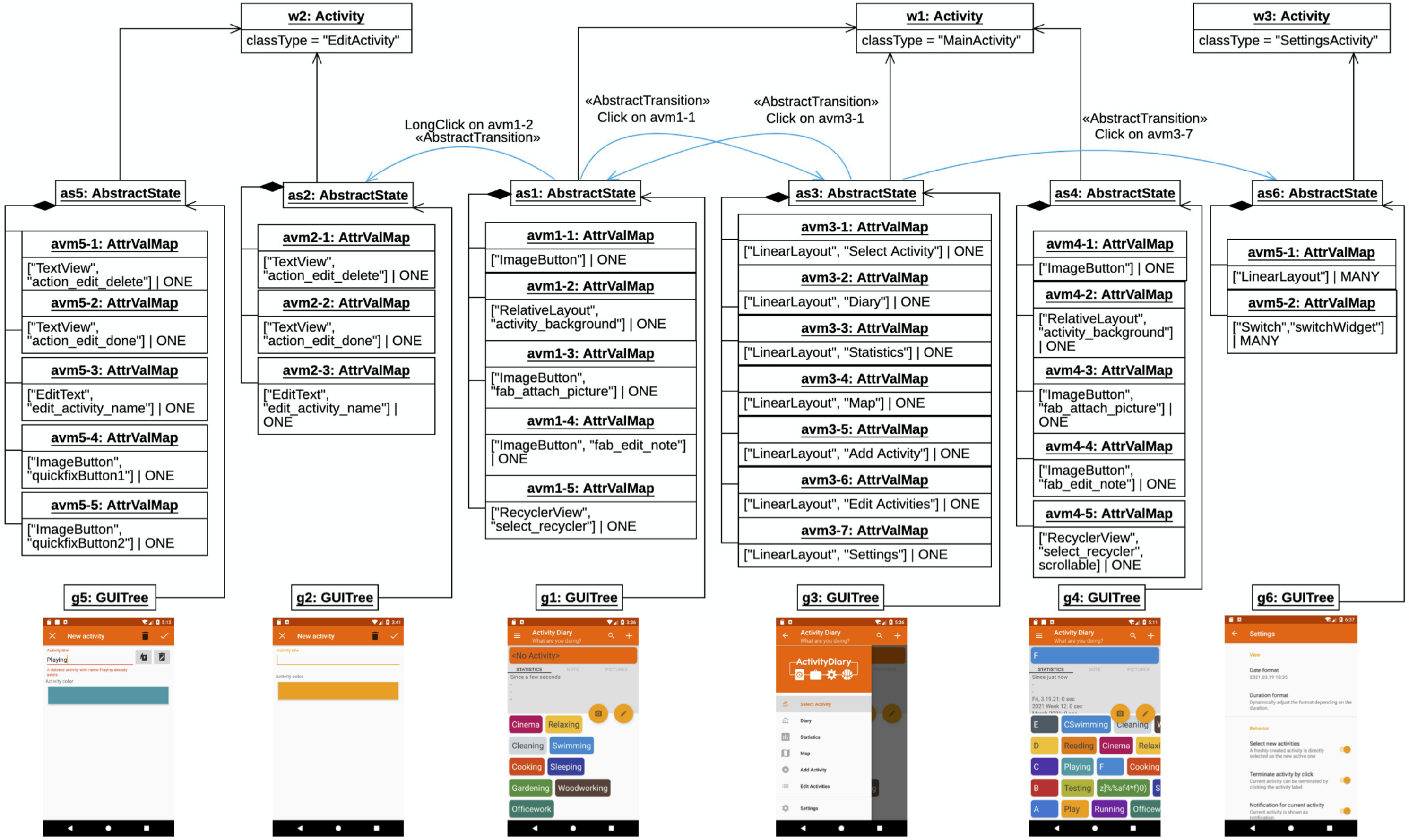


Figure 3.2: ATUA Metamodel. Colors are used to identify classes belonging to a specific metamodel component: light blue for EWTG (bottom), light green for DSTG (middle), orange for GSTG (top).


w2: Activity
classType = "EditActivity"
w1: Activity
classType = "MainActivity"
w3: Activity
classType = "SettingsActivity"
LongClick on avm1-2
«AbstractTransition»
«AbstractTransition»
Click on avm1-1
«AbstractTransition»
Click on avm3-1
«AbstractTransition»
Click on avm3-7
as5: AbstractState
avm5-1: AttrValMap
["TextView", "action_edit_delete"] | ONE
avm5-2: AttrValMap
["TextView", "action_edit_done"] | ONE
avm5-3: AttrValMap
["EditText", "edit_activity_name"] | ONE
avm5-4: AttrValMap
["ImageButton", "quickfixButton1"] | ONE
avm5-5: AttrValMap
["ImageButton", "quickfixButton2"] | ONE
as2: AbstractState
avm2-1: AttrValMap
["TextView", "action_edit_delete"] | ONE
avm2-2: AttrValMap
["TextView", "action_edit_done"] | ONE
avm2-3: AttrValMap
["EditText", "edit_activity_name"] | ONE
as1: AbstractState
avm1-1: AttrValMap
["ImageButton"] | ONE
avm1-2: AttrValMap
["RelativeLayout", "activity_background"] | ONE
avm1-3: AttrValMap
["ImageButton", "fab_attach_picture"] | ONE
avm1-4: AttrValMap
["ImageButton", "fab_edit_note"] | ONE
avm1-5: AttrValMap
["RecyclerView", "select_recycler"] | ONE
as3: AbstractState
avm3-1: AttrValMap
["LinearLayout", "Select Activity"] | ONE
avm3-2: AttrValMap
["LinearLayout", "Diary"] | ONE
avm3-3: AttrValMap
["LinearLayout", "Statistics"] | ONE
avm3-4: AttrValMap
["LinearLayout", "Map"] | ONE
avm3-5: AttrValMap
["LinearLayout", "Add Activity"] | ONE
avm3-6: AttrValMap
["LinearLayout", "Edit Activities"] | ONE
avm3-7: AttrValMap
["LinearLayout", "Settings"] | ONE
as4: AbstractState
avm4-1: AttrValMap
["ImageButton"] | ONE
avm4-2: AttrValMap
["RelativeLayout", "activity_background"] | ONE
avm4-3: AttrValMap
["ImageButton", "fab_attach_picture"] | ONE
avm4-4: AttrValMap
["ImageButton", "fab_edit_note"] | ONE
avm4-5: AttrValMap
["RecyclerView", "select_recycler", scrollable] | ONE
as6: AbstractState
avm5-1: AttrValMap
["LinearLayout"] | MANY
avm5-2: AttrValMap
["Switch","switchWidget"] | MANY
g5: GUITree
g2: GUITree
g1: GUITree
g3: GUITree
g4: GUITree
g6: GUITree
New activity
Activity Diary
Settings

Legend: Straight, black arrows capture associations. To avoid cluttering, we rely on curved, blue arrows to model AbstractTransitions. The action and target of abstract transitions are reported in textual form. For illustration purposes, we show screenshots instead of GUITree hashes. Cardinality of AttrbuteValuationMaps are reported after the "|" symbol.

Figure 3.3: Example of an App model, represented as a UML object diagram.

target of any action of type WidgetActionType (See Figure 3.2).

At runtime, during testing, ATUA identifies AbstractStates through a dedicated *abstraction function* ($\mathcal{L}$). ATUA automatically defines a distinct $\mathcal{L}$ for each Window of the AUT. $\mathcal{L}$ relies on a predefined set of *reducers*, i.e., functions that extract the value of a property of a widget [37]. Table 3.1 shows the list of reducers implemented by ATUA. Two AbstractStates differ when at least one value differs across their respective AttributeValuationMaps, or when they have a different cardinality. For example, in Figure 3.3, the AbstractStates *as1* and *as3* differ because *as1* does not contain the AttributeValuationMaps for the LinearLayouts belonging to the drawer menu (to save space, we do not report all the AttributeValuationMaps for *as3*). The AbstractStates *as1* and *as4* are different because in *as4* the RecyclerView (avm4-5) becomes scrollable.

In the DSTG, state transitions are captured by AbstractTransitions. An AbstractTransition is univocally identified by the *actionType*, its *source*, its *target*, its *destination*, and its *actionData*. The *actionType* matches one of the items belonging to the enumerations WidgetActionType or WindowActionType. The *actionData* is captured only for two actionTypes (i.e., Swipe and Intent) that usually lead to distinct AbstractStates depending on their action data. In the case of Swipe, the actionData indicates the direction of the Swipe action (i.e., Up, Down, Left, Right). For Intents, the actionData matches the Intent input text because we expect engineers to provide one manual input for each possible Intent type (e.g., one different URL for each of the file types supported by an App).

A DSTG may include non-deterministic AbstractTransitions, that is, transitions with the same actionType, outgoing from the same AbstractState, but reaching different AbstractStates. Non-deterministic AbstractTransitions may prevent us from finding the correct sequence of Inputs necessary to reach the states in which target methods could be triggered. ATUA detects non-determinism at runtime, during testing, when an Action does not bring the App into the expected AbstractState. When this happens, ATUA refines $\mathcal{L}$ for the AbstractState in which the action had been triggered. It does so according to five levels of granularity, which are captured in Table 3.2. With level L1, $\mathcal{L}$ distinguishes states based on static information about the widgets (i.e., resource ID and class) and information about how they can be interacted with (i.e., reducers appearing in rows 3 to 12 of Table 3.1). With level L2, in addition to the information accounted for in L1, $\mathcal{L}$ includes the text associated to the widget, which

Table 3.1: ATUA reducers. We indicate the value of the *property* reported by each.

	Reducer	Description
1	R_{RID}	Resource ID.
2	R_{CN}	Class name.
3	R_{CD}	Value of *Content description*.
4	R_P	Value of *Password*.
5	R_C	Value of *Clickable*.
6	R_{LC}	Value of *Long Clickable*.
7	R_S	Value of *Scrollable*.
8	R_{Ch}	Value of *Checked*.
9	R_E	Value of *Enabled*.
10	R_S	Value of *Selected*.
11	R_I	True if it is an input field.
12	R_T	Value of *Text*.
13	R_{HC}	True if the widget contains one or more children.

Table 3.2: Refinement of ATUA state abstraction function. In blue we show the reducers introduced in finer granularity level.

Level	Reducers applied to interactable Widget	Reducers applied to interactable Widget Children
L1	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S$	
L2	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T$	
L3	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T, R_{HC}$	
L4	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T$	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S$
L5	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T$	$R_{RID}, R_{CN}, R_{CD}, R_{Ch}, R_E,$ $R_P, R_S, R_I, R_C, R_{LC}, R_S, R_T$

often affects the behaviour of an app (e.g., invalid characters in a textbox may prevent a state transition). With level L3, $\mathcal{L}$ also reports the number of children of a widget (i.e., R_{HC}). L3 is useful because the interactive widgets captured by $\mathcal{L}$ may include non-interactive children whose state is not captured by $\mathcal{L}$ but may characterize the current state (e.g, through descriptive labels). With levels L4 and L5, $\mathcal{L}$ captures, for every interactive widget, the same information as L2 and, in addition, for every child, the information captured by levels L1 and L2, respectively.

Figure 3.4 shows the result of the refinement of $\mathcal{L}$ for the abstractState *as3*. By applying $\mathcal{L}$ with level L1, all the clickable elements on the Window, which are LinearLayouts with the same properties, except for the text, resulted into a same AttributeValuationMap with cardinality *MANY*. At runtime, ATUA detects non-determinism; indeed, a click on this AttributeValuationMap may lead to two different AbstractStates: *as1* and *as6*. The refinement of $\mathcal{L}$, which leads to level L2, allows ATUA to distinguish all the different clickable elements since the text property is included into the AttributeValuationMap, thus eliminating non-determinism.

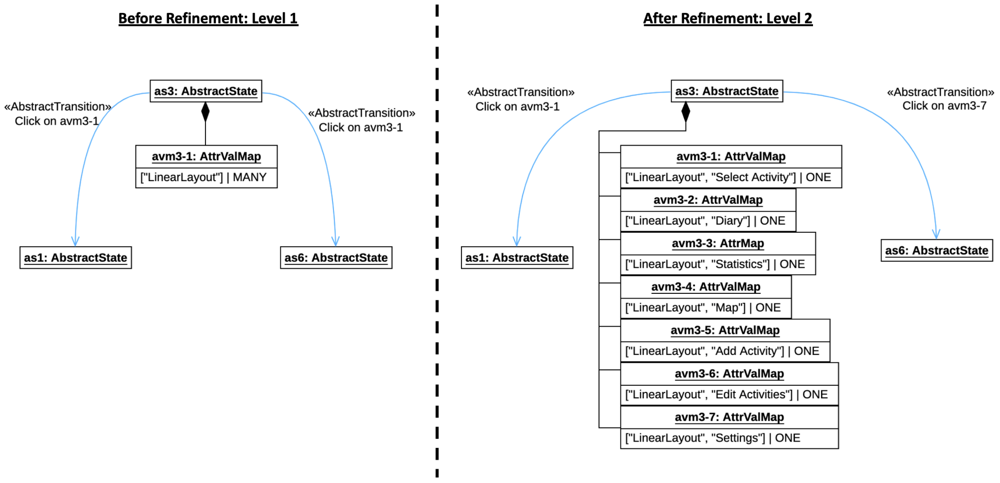


Figure 3.4: Example of refinement of $\mathcal{L}$. For the notation used, see the Legend of Figure 3.3.

3.2.2 Step 1. Identify Updated Methods

In the first step, ATUA identifies methods that have been modified or introduced by the new version of the App (i.e., V2 in Figure 3.1). This task might be accomplished through source code comparison across versions [87]. However, to enable experiments with commercial Apps, we developed a toolset (hereafter, AppDiff) that compares compiled Android Apps.

AppDiff is an extension of *LibScout* [88], a light-weight static analysis tool for Android. It first generates a hashtree over the bytecode for each App. The hashtree is a three-layered Merkle tree in which parent hashes are generated from their child nodes. The three layers model the flattened package structure that is preserved in the compiled code, i.e., packages, classes, and methods.

The tree is built bottom up starting with the method hashes. A method hash is computed over the method signature and the opcodes in bytecode instructions. To identify code-level changes across App versions, we additionally store package, class, and method names along with the hashes. To efficiently check for differences, two hash trees are matched top-down starting with the package hashes. Methods that share the same name but have a different hash have been modified. New methods appear only in the most recent version.

3.2.3 Step 2. Generate the Extended WTG

ATUA generates the Extended WTG by means of static program analysis; more precisely, by performing, on the updated App, the analysis implemented by an extended version of

GATOR and Soot.

The original version of GATOR works by processing Android bytecode and XML layout files. For the analysis of bytecode, GATOR relies on Soot. Bytecode analysis is used to identify the types of Window (i.e., Activity, Dialog, OptionsMenu, and ContextMenu) that are programmatically specified in the App. Bytecode analysis is also used to identify the widgets that compose a window and the associated event handlers. GATOR identifies widgets that extend the class *android.view.View* and its handlers. Event handlers' code is processed to determine window transitions. XML layout files are processed to identify additional event handlers.

Our extensions to GATOR address some known limitations [89]. More precisely, we support the identification of window transitions triggered by Fragments and RecyclerView, which are widget containers that are not identified by GATOR as such. Our extensions associate the contained widgets to the window that declares either the Fragment or the RecyclerView. Also, in the EWTG, we associate each WindowTransition with the Input triggering the transition (the information is provided by GATOR).

We rely on Soot to traverse the backward call graph of every updated method m. During the traversal, when we encounter a method that has been identified by GATOR as an event handler e, we update the EWTG to trace the fact that the inputs associated to the WindowTransition triggered by the event handler e can lead to the execution of the updated method m (i.e., we add the updated method m to the list of target methods for the Input instance). Also, we rely on Soot to extract string literals to be used for testing (see Section 3.2.5).

Finally, we determine if GATOR does not identify some of the event handlers of the App, which is a common problem of static analysis tools for Apps. Indeed, these static analysis tools rely on hardcoded procedures for the identification of event handlers (e.g., they look for specific method names [85]); since OS APIs are under continuous evolution, it is unlikely that static analysis tools will ever be able to identify all the event handlers of an App. To address this problem, we introduced into ATUA three solutions, one based on static analysis (described in the next paragraph) and two based on dynamic program analysis (described in Sections 3.2.5 and 3.2.5).

To identify missing event handlers using static analysis, we rely on the observation that if an event handler is not detected by GATOR, the backward traversal of the call graph

```
{
  "BookInsertionAndSearch" : {          //input pattern
    "Windows" : [
        "ACT[bookcatalogue.EditAuthorList]1741",
        "ACT[bookcatalogue.BookEdit]1802,
        "ACT[bookcatalogue.BookISBNSearch
        ]1843" ],
    "DataFields" : {
      "isbn" : {
        "resourceIdPatterns" : [ "isbn_txt" ]
      },
      "title" : {
        "resourceIdPatterns" : [ "title_txt" ]
      },
...
    },
    "Instances" : [
      {
        "isbn" : "0387284540",
        "title" : "Applied probability and statistics",
        "publisher" : "Springer",
        "pages" : "350",
        "list_prices" : "69",
        "format" : "Hard Cover",
        "genre" : "Unfiction",
        "language" : "English"
      }
```

Figure 3.5: Manual definition of inputs

performed by ATUA will not reach any event handler but will terminate in a method that (i) belongs to a Window class and (ii) is not invoked by any other method of the AUT. Such methods are likely event handlers invoked at runtime by the Android APIs. We refer to them as *hidden-handlers*. We keep track of all the hidden-handlers encountered during the analysis along with the list of updated methods reachable from them. We rely on this information during Step 5.

3.2.4 Step 4. Prepare manual inputs

Certain input values are unlikely to be automatically generated, consequently certain Apps' features might not be automatically exercised without an appropriate solution to handle such cases. In related work, these inputs are referred to as *Unlocking GUI Input Event Sequences* (hereafter, *unlocking inputs*, for brevity) [90]. Examples include login credentials, files of a specific type, and data to be received by the AUT through the Android Intent mechanism. To handle these cases, in ATUA, engineers specify unlocking inputs in a JSON file capturing a set of input values and the patterns identifying the Windows requiring such input values. An example of our input definition format is provided in Figure ??.

According to our format, engineers can specify one or more input insertion patterns (e.g., *BookInsertionAndSearch* in Figure ??, Line 2). For each pattern, they specify the Windows in which the pattern should be used (Line 3), and the widgets which should be used to provide the input data specified (i.e., *DataFields* field in Line 4). Each widget is identified by a name (e.g., *isbn* in Line 5) and a regular expression that enables its selection in the GUITree, based on its name (e.g., *isbn_txt* in Line 6). Finally, multiple input instances (e.g., book names in this case) can be specified (see field *Instances* in line 13).

Since ATUA relies on software engineers to identify unlocking inputs, its effectiveness might be affected by engineers' mistakes. For example, in our experiments, we specify manual inputs only for login operations and key features on the AUT (see Section 3.4.2), thus potentially omitting unlocking inputs concerning other App features.

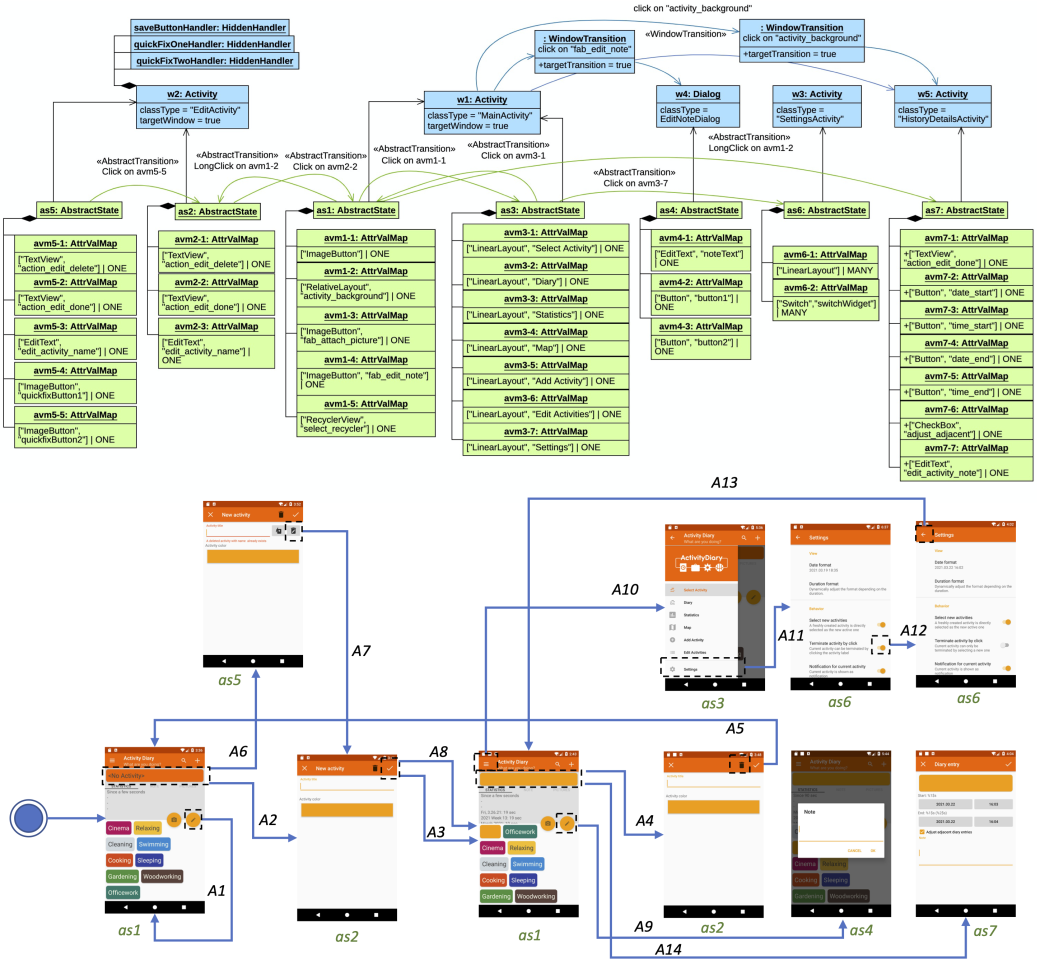
3.2.5 Step 5. Automatically test the App

ATUA automatically tests the updated App by triggering the Actions required to exercise target Transitions. When testing starts, the App model consists of an instance of the EWTG for the AUT. GSTG and DSTG are dynamically constructed and extended at runtime by ATUA.

The test execution process includes three distinct phases, each one relying on a different strategy for the generation of Actions. In *Phase 1*, ATUA triggers one Action for every target Input. The goal of Phase 1 is to handle the simplest scenario, i.e., exercise instructions that are executed every time data provided through a target Input is processed. In *Phase 2*, ATUA exercises target Windows with multiple, diverse sets of Inputs. The goal of Phase 2 is to exercise those instructions that are executed only when specific constraints on input values provided in a Window are satisfied. In *Phase 3*, ATUA exercises both target Windows and Windows they depend on. The goal of Phase 3 is to exercise those instructions that can be executed only when certain constraints on the input values provided in related Windows (e.g., preferences Windows) are satisfied.

Running Example

To illustrate our approach, we describe part of the actions taken by ATUA when testing the upgrade to version 134 of Activity Diary. Activity Diary enables end-users to record a diary



Legend: The top part shows the EWTG, the DSTG is in the middle, while, to simplify reading, the GSTG is represented by means of screenshots corresponding to each GUITree. Labels below screenshots are used to associate GSTG states to AbstractStates. The Actions appearing in the GSTG are: *A1*, click on *EditNote* button. *A2*, long click on *Current activity* widget. *A3*, click on *Save* button. *A4*, long click on *Current activity* widget. *A5*, click on *Delete activity* button. *A6*, long click on *Current activity* widget. *A7*, click on the button to automatically rename the deleted activity. *A8*, click on *Save* button. *A9*, click on *EditNote* button. *A10*, click on *Open navigation* button. *A11*, click on *Settings* button. *A12*, click on *Terminate activity by click* button. *A13*, click on *Back* button. *A14*, click on *Current activity* widget.

Figure 3.6: App Model for the Activity Diary running example

for their activities. It includes features to categorize activities, report statistics, remind users about recurrent activities, and attach notes and pictures to activities.

We consider a subset of the features updated in version 134 of Activity Diary, which aim to (1) visualize the details of the current activity by clicking on the activity name, (2) edit the current activity or create a new activity with a long click on the current activity name, (3) automatically fix a duplicated name for an activity, (4) edit an activity note. Figure 3.6 shows the App model for our running example.

In the EWTG of Figure 3.6, the transition between MainActivity (*w1*) and EditNote-Dialog (*w4*) is a target transition since it triggers one modified method: the handler of the

EditNote button (i.e., *editNoteHandler*, not shown in Figure 3.6). The EditActivity Window (*w2*) is a target Window since it contains three hidden-handlers: *saveButtonHandler*, *quickFixHandlerOne*, and *quickFixHandlerTwo*. The hidden-handler *saveButtonHandler* reaches the modified method *checkConstraints* while the other two hidden-handlers are the event handlers for the quick fix buttons appearing in the UI. ATUA classifies these three methods (i.e., *saveButtonHandler*, *quickFixHandlerOne*, and *quickFixHandlerTwo*) as hidden-handlers because they are not associated by GATOR to any WindowTransition in the EWTG (see Section 3.2.3). Indeed, GATOR cannot correctly process the control flow that reaches function *setListener*, which is the function used to assign the three handlers to their corresponding buttons[3].

The GSTG in Figure 3.6 captures the sequence of Actions triggered by ATUA during testing[4]. They are described in the following sections, where appropriate.

Detection of the active Window

A building block of our test automation strategy is the detection of the active Window. More precisely, ATUA should determine if a pop-up (i.e., a Dialog, an OptionsMenu, or a ContextMenu) is open on top of the active Window. Neither Android nor DM2 provides such information. To determine if a pop-up is open, ATUA relies on the dimensions of the active Window on the screen. Indeed, if an Activity is displayed, its dimensions should match the dimensions of the screen. Otherwise, the currently displayed Window is either a Dialog, an OptionsMenu, or a ContextMenu. To determine which Dialog or Menu is open, ATUA identifies the Window of the EWTG with the highest portion of Widgets visualized on the screen. ATUA computes the ratio, R_w, of Widgets belonging to Window w that appear in the displayed GUITree. More precisely, ATUA computes R_w for all the Dialogs and Menus that can be triggered by the current Window. The active Window is the one with the highest values for R_w.

If, due to the limitations described in Section 3.2.3, static analysis does not detect that, for a certain Window w, there is an event handler that will pop-up a certain Dialog or Menu p, ATUA may not be able to find a Dialog or Menu to be matched with the displayed GUITree.

[3] GATOR does not correctly process control flows starting within event handlers, likely because of their recursive nature; in our running example, the control flow starts within event handlers triggered by changes in color selectors and text boxes.

[4] A demo video for the running example is available online [91]

More precisely, ATUA may observe that (1) the current Window has no Dialogs and Menus associated to it in the EWTG or (2) the score computed for every Dialog and Menu of the current Window is zero. To overcome such a problem, in these scenarios, ATUA updates the EWTG to include a new pop-up Window.

ATUA testing algorithm

At runtime, after detecting the active Window, ATUA automatically derives the current AbstractState according to the procedure described in Section 3.2.1. The subsequent activities depend on the current testing phase.

The activities performed in the three phases follow the same algorithm, which is presented in Algorithm 1. What differentiates the three phases are the strategies adopted to exercise the App and the test budget allocated . Line 1 in Algorithm 1 shows that the algorithm iterates till the test budget for the current phase is consumed (function *phaseBudgetConsumed*), all the targets for the current phase have been covered (function *coverageTargetsExercised*), or it cannot further improve coverage (function *stagnation*).

The iteration starts by identifying a test target (function *selectTarget*, Line 3). The test target is either a Window or a WindowTransition. A new test target is identified when no target has been selected yet or the current target has already been fully exercised (Line 2). After identifying the test target, ATUA relies on the App model to identify the test target path (Line 4), i.e., a sequence of Actions that makes the App render the target Window or reach the target AbstractState.

The test target path is derived with a breadth-first traversal of the App model. The traversal starts from the current AbstractState. The traversal proceeds through both AbstractTransitions and WindowTransitions. A WindowTransition is taken only if an AbstractTransition is not available. The visit of the model stops when we reach the test target or all the reachable nodes are explored.

So long as a test target is not reached (Line 7), ATUA executes function *reachTargetNode* (Line 8), which triggers the next Action in the test target path. For each Action in the test target path, we know the Window or the AbstractState to expect. After executing an Action, function *reachTargetNode* checks if the App is in the expected Window or AbstractState. If not, function *reachTargetNode* flags the target as not reached, and returns to the main

Algorithm 1 ATUA testing algorithm.

```
 1: while ( NOT stagnation() ) AND ( NOT phaseBudgetConsumed( phaseBudget) ) AND (NOT coverageTargetsExercised() ) do
 2:     if target not selected OR target already exercised OR visitBudget exhausted then
 3:         selectTarget()
 4:         identifyPathToTarget()
 5:     else if target unreachable then
 6:         identifyPathToTarget()
 7:     if NOT targetReached() then
 8:         reachTargetNode( reachabilityBudget )
 9:     else
10:         exerciseTarget( targetBudget )
11:     if additional random exploration required then
12:         performRandomExploration( randomExplorationBudget )
```

execution loop to look for a different path to reach the test target (Line 5). When a target is reached (Line 9), ATUA exercises the target according to the Action generation strategy for the current phase (function *exerciseTarget*).

Finally, random exploration of the active Window might be triggered by functions *reachTargetNode* and *identifyPathToTarget* to improve the EWTG (Line 12). This is described in Section 3.2.5.

To regulate the allocation of the phase budget (i.e., how many Actions each function invoked by the algorithm is allowed to generate), the ATUA algorithm makes use of three budget variables: (1) *reachabilityBudget*, which specifies the maximum number of Actions to be used to reach a target node, (2) *targetBudget*, which specifies the number of Actions to be used to exercise the target node, (3) *randomBudget*, which specifies the number of Actions to be used for random exploration. At runtime, when counting the number of Actions performed, we ignore Actions of type TextInput and Click on checkboxes since they generally do not trigger WindowTransitions. The budget variables are initialized with different values depending on the current test phase. Table 3.3 provides an overview of the criteria adopted, which are described in details in the following paragraphs. To define budgets, a *scale factor* is used to optimally distribute the test budget across phases and test targets. For example, with a test budget of five hours, we can invest more time in Phase 2 than with a test budget of one hour.

Random exploration

Functions *reachTargetNode* and *identifyPathToTarget* in Algorithm 1 may trigger the random exploration of the AUT to improve the EWTG. This is done when Inputs cannot be exercised

Table 3.3: Strategies adopted, in different ATUA phases, to define the budget allocated to distinct test activities.

Phase	Budget	Strategy
Phase1	Phase	Infinite, i.e., all the target windows are exercised till stagnation is detected or all the targets are covered.
	Reachability	Infinite, i.e., all the paths are traversed in this phase.
	Target	Infinite, i.e., all the target Inputs are tried in this phase.
	Random Exploration	Set to $\mathit{scaleFactor} \cdot \mathit{NumberOfActionsForActiveWindow}$. $\mathit{NumberOfActionsForActiveWindow}$ is the number of distinct Actions that can be performed in the active window; it is based on the interactive information associated to a widget (e.g., we perform a Click Action if the widget is clickable, or four Swipe Actions - one for each swipe direction - if it is scrollable). In our experiments, we set $\mathit{scaleFactor}$ to 1 for an overall test budget of one hour, to 2 for a test budget of five hours. Random exploration is triggered by either *reachTargetNode* or *identifyPathToTarget*.
Phase2	Phase	Set to $\mathit{scaleFactor} \cdot \mathit{NumberOfTargetWindows}$.
	Reachability	Set to $\mathit{scaleFactor} \cdot \mathit{actionsThreshold}$. The value is reset every time a new TargetWindow is identified. We set $\mathit{actionsThreshold}$ to 25.
	Target	Set to be equal to what remains of the ReachabilityBudget after the target window is reached. In other words, $\mathit{ReachabilityBudget} + \mathit{TargetBudget} = \mathit{scaleFactor} * \mathit{actionsThreshold}$
	Random Exploration	Set to $\mathit{scaleFactor} \cdot \mathit{randomThreshold}$. Random exploration is triggered by either *reachTargetNode* or *identifyPathToTarget*. We set $\mathit{randomThreshold}$ equal to $\mathit{actionsThreshold}$.
Phase3	Reachability	Not used in this phase.
	Target	Set to $\mathit{scaleFactor} \cdot \mathit{actionsThreshold}$. It is reset every time a new TargetEvent is identified.
	Random Exploration	Set to $\mathit{scaleFactor} \cdot \mathit{actionsThreshold}$. Random exploration is triggered (1) to explore the related Window, (2) when the related Window cannot be reached through the identified path, (3) when the target Window cannot be reached through the identified path (see Section 3.2.5). We set $\mathit{randomThreshold}$ to 5.

and a test target cannot be reached.

Function *reachTargetNode* determines that an Input cannot be exercised when the associated Widget is not visible or enabled in the GUITree. It happens, for example, when the content of a *NavigationDrawer* varies based on the buttons pressed in the active Window. To make the required Widget visible, function *reachTargetNode* randomly exercises the active Window. This is done by iteratively and randomly selecting one widget among the ones that have been exercised less frequently in the active Window. The selected widget is then exercised according to the strategies listed in Table 3.4. The exploration of the active Window terminates when the desired widget is found or when the test budget for random

Table 3.4: ATUA Input generation procedures.

Widget type	Input generation procedure
Any widget	Trigger an InputEvent among the ones for which an event handler has been declared.
Textarea	Randomly apply one of the following: (1) leave it empty, (2) reuse a string already used in the past, (3) reuse a string already used for the same widget, (4) reuse a string literal extracted with static analysis, (5) use a randomly generated alphabetic string [43], (6) use a randomly generated non alphabetic char.
Radio buttons and check boxes	Randomly select one of the possible options (e.g., checked/not checked, for check boxes).
Widgets with manual input	Randomly select one of the available InputInstances, if more than one is available, and then assign the specified value.
Intent	If the current activity declares an Intent, it triggers the Intent specified by the engineer.

exploration is exhausted.

Function *identifyPathToTarget* may determine that it is not possible to find a path to a test target. This happens when the EWTG does not include all the WindowTransitions, which is due to the limitations of static analysis tools mentioned in Section 3.2.3. For example, GATOR does not detect the Animation design pattern [92], which leads to a WindowTransition. When a test target path is not found, ATUA performs a random exploration of the active Window and then resumes the execution from the beginning of the main execution loop. ATUA records of all the unreachable targets identified.

Phase 1

In Phase 1, a test target is any Window with target Inputs that have not been triggered yet. Function *selectTarget* randomly selects a Window with such characteristics (Line 3 in Algorithm 1). After selecting the target Window, ATUA follows the path to reach the test target.

Function *exerciseTarget*, first produces *user-like inputs* (i.e., input values for text areas, radio buttons, and check boxes), as specified in Table 3.4. Then it triggers an Action that exercises a randomly selected target Input. Function *exerciseTarget* keeps triggering Actions that exercise target Inputs until all the target Inputs have been exercised or another Window has been visualized. ATUA then resumes the execution of the main loop (i.e., Line 1 in Algorithm 1). When the target Window is associated to hidden-handlers that can reach target methods, ATUA also performs a random exploration of the Window. If an Action triggers the execution of an hidden-handler, ATUA introduces a corresponding WindowTransition into the EWTG.

To maximize the chances of exercising every target Input, which is the objective of Phase 1, the phase, reachability, and target budgets are infinite. More precisely, we try to reach every TargetWindow (infinite *phaseBudget*) by trying every possible path (infinite *reachabilityBudget*); furthermore, we exercise every TargetWindow with all the target Inputs (infinite *targetBudget*).

In Phase 1, we observe *stagnation* when all the remaining targets either cannot be reached or all their target Inputs cannot be exercised.

Running Example. In Phase 1, ATUA triggers one Action for every target input. By default, Activity Diary starts by rendering the MainActivity with a predefined set of activities and no current activity being selected. Since MainActivity is a target Window, ATUA exercises it. First ATUA clicks on the edit note button (Action A1 in Figure 3.6) and, since there is no current activity selected, Action A1 partially covers the target methods (indeed, Activity Diary does not open the EditNote Window, which will be achieved in Phase 2). Then, ATUA triggers a long click on the current activity widget (A2), which leads to an instance of the EditActivity Window. Since EditActivity is a target Window, ATUA aims to exercise it. However, EditActivity contains only hidden-handlers, not target Transitions. For this reason, ATUA performs a random exploration of the window; during the random exploration, after filling the window with random inputs, ATUA clicks on the save button (A3), which triggers the execution of one of the updated methods (i.e., *checkConstraints*) and thus ATUA introduces into the DSTG a new AbstractTransition associated to the updated method (i.e., the abstract transition between *avm2-2* and *as1* in Figure 3.6, which does not have a corresponding WindowTransition in the EWTG). After these three actions, ATUA has exercised both the MainActivity and the EditActivity and can thus move to Phase 2. In Phase 1, ATUA has thus exercised all the easy-to-reach target methods in a few steps.

Phase 2

In Phase 2, we aim to maximize the coverage of those target methods that have not been fully covered. To this end, the target Window shall be the one that can trigger the execution of the highest number of uncovered instructions. Also, since the AbstractState of an App might affect the reachability of a target method, we should give higher priority to Windows with target methods exercised in fewer AbstractTransitions.

To achieve the above-mentioned objectives, we select as target Window the one that maximizes the score WS_w,

$$WS_w = \sum_{m \epsilon MT_w} c_w \cdot u_m$$

where MT_w is the set of target methods associated to the target Inputs of Window w. Term u_m is the number of uncovered instructions belonging to method m. A target Window x can thus be any Window with $WS_w > 0$. A Window w is selected as target with a probability proportional to WS_w. In the formula above, c_w is a weight introduced to focus first on those methods that have been covered less. It is the complement of the proportion of AbstractTransitions that exercise the method:

$$c_m = 1 - \frac{AA_m}{AA}$$

where AA_m is the number of AbstractTransitions that covered method m and AA is the number of AbstractTransitions in the App model.

To select the test target path, ATUA identifies the AbstractState with the highest number of uncovered instructions belonging to interactive widgets, which is captured by the AS_{as_w} score:

$$AS_{as_w} = \sum_{m \epsilon MT_{as_w}} c_w \cdot u_m$$

where as_w is an AbstractState for the Window w, and MT_{as_w} is the set of target methods that might be covered through as_w. The set MT_{as_w} consists of all target methods triggered by either (1) an Intent or (2) an InputEvent for an interactive widget in as_w.

Function *exerciseTarget* works in the same way as in Phase 1. However, Phase 2 differs from Phase 1 with respect to the target, phase, and reachability budgets. Indeed, to uniformly distribute the phase budget across the selected target Windows, in Phase 2, the budget for reaching a test target and exercising it is set to "$scaleFactor \cdot actionsThreshold$", with $actionsThreshold$ representing a number of Actions that, based on preliminary experiments, is sufficient to reach a target and exercise it (in our experiments, we set its value to 25). In Phase 2, the *phase budget* is exhausted when ATUA has exercised a number of windows that is equal to "*scaleFactor· overall number of TargetWindows*". Also, a same Window can be selected as a target multiple times. By repeatedly generating different sets of Actions for a same Window, ATUA covers different combinations of user-like inputs, which may include combinations that lead to the coverage of different sets of instructions.

In Phase 2, we observe *stagnation* when, after exercising all the available targets, the coverage of target methods has not increased.

Running Example. Phase 2 is necessary to maximize the coverage of updated methods reached through the MainActivity and the EditActivity Windows. The target Window with the highest WS_w score is EditActivity because some of the instructions of method *checkConstraints* and all the instructions implementing the quick fix feature have not been exercised in Phase 1. MainActivity has a lower WS_w score since only a few instructions of the EditNote button handler are not covered.

ATUA selects the EditActivity Window as first target; at this stage, it has only one AbstractState (i.e., *as2* in in Figure 3.6). ATUA reaches the EditActivity Window with a long click (Action A4) on the the current activity (an activity with an empty name). EditActivity includes one target AbstractTransition, the one exercising *checkConstraints*. While generating inputs to maximize the coverage of method *checkConstraints*, ATUA clicks on the button that deletes the activity and brings the App back to the MainActivity (A5). From the main Activity, ATUA performs again a long click on the currentActivity widget (A6), which leads to an instance of EditActivity for the definition of a new activity where the quick fix buttons are visualized. In this case, the quick fix buttons are visualized because an activity with an empty name (the default for new activities) had ben selected and Activity Diary already contains an activity with an empty name (i.e., the one deleted by Action A5). Because of the presence of the two quick fix buttons, ATUA introduces a new abstract state into the DSTG (i.e., *as5*).

To cover the hidden-handlers of EditActivity, ATUA exercises the quick fix button that automatically renames the activity having a conflicting name (A7). Finally, ATUA selects MainActivity as target and exercises the EditNote button (A9), which pops-up the EditNote Dialog thus covering the missing lines. In Phase 2, ATUA successfully covered all the target methods triggered within a target Window (i.e., EditActivity), by repeatedly exercising it.

Phase 3

In Phase 3, we aim to cover those target method instructions that exhibit data dependencies from state variables defined by Windows different than the one reaching a target method. Examples include instructions that can be executed only after enabling specific options in the preferences Window of the App. For this reason, in Phase 3, the test target is a WindowTransition presenting associated targetMethods that remain to be fully covered. Also, for each WindowTransition to be tested, we need to identify a set of related Windows that should be exercised before executing it.

Function *selectTarget* returns a WindowTransition belonging to a target Window selected according to the same criteria as for Phase 2, i.e., with a probability proportional to its WS score. To minimize the effort spent in reaching target Windows, once a target Window has been selected, function *selectTarget* iteratively returns each target WindowTransition belonging to it.

When a test target has been selected, in function *exerciseTarget*, ATUA (1) identifies the related Window that should be exercised first, (2) identifies a path to this related Window, (3) reaches the related Window and randomly exercises it, (4) identifies a path to the closer AbstractState for the target Window in which the target WindowTransition is enabled, and (5) reaches the identified AbstractState and triggers a target Input. In Phase 3, function *identifyPathToTarget* is not invoked because testing starts from the related Window. Consistent with Phase 2, the *targetBudget* is set to $scaleFactor \cdot actionsThreshold$.

Function *exerciseTarget* relies on random exploration (1) to explore the related Window, (2) when the related Window cannot be reached through the identified path, (3) when the target Window cannot be reached through the identified path. The random exploration budget is set to $scaleFactor \cdot randomThreshold$. Since random exploration has been largely used in previous phases and to limit the time spent in related windows, in Phase 3, we set $randomThreshold$ to a value lower than the one used in Phase 2 (e.g., we used 5 in our experiments).

To identify related Windows, we rely on information retrieval techniques. We do not rely on traditional data-flow analysis [59] because data dependencies might be implemented in many different forms (e.g., setting a state variable in a shared object or saving a property in a key-value registry) that are not fully identified by such analysis.

Related Windows are retrieved through the computation of the term frequency (TF) and inverse document frequency (IDF) metrics, which are standard information retrieval metrics [93]. In the following, we discuss how we compute these metrics.

Since dependencies between Windows are due to either state variables defined in shared objects or property values in key-value registries, the executable code of Windows presenting such dependencies should share a subset of *class attributes* and *literals*. For this reason, the terms used to identify dependencies are *class attributes* and *literals* appearing in the implementation of the methods of the App (extracted with Soot).

We compute $TF(t, h)$, the frequency of the term t for an Input handler h, as the number of methods in which the term appears, considering the handler itself and any of the methods invoked by the handler. To include only terms that characterize the functionality triggered by the WindowTransition, we consider only methods declared in the same class of the handler or in its inner or outer classes.

The frequency of the term t for a WindowTransition wt is computed as the sum of the term frequency for all the handlers of the Input (HI_{wt}) that triggers the transition,

$$TF(t, wt) = \sum^{h \in HI_{wt}} TF(t, h)$$

The frequency of the term t for a Window w, $TF(t, w)$, instead, is computed as the number of methods in which the term appears, considering the methods that are either declared in the class that implements the Window or in its parent class.

The inverse document frequency of a term is computed as

$$IDF(t) = log(\frac{total\ number\ of\ Windows}{number\ of\ Windows\ in\ which\ t\ appears})$$

The related Windows for a WindowTransition wt can be identified by computing a dependency score for every Window w of the App, as follows

$$DS(w, wt) = \sum^{t \in T} NW(t, w) \cdot NW(t, wt)$$

where T is the set of terms for the App, and $NW(t, d)$ is the normalized term weight, which captures the extent to which a term is representative for either a Window or a WindowTransition. $NW(t, d)$ is computed according to a standard formula [93]:

$$NW(t, d) = \frac{TF(t, d) \cdot IDF(t)}{EL(d)}$$

where EL(d) is the Euclidean Length of the element d (i.e., a Window or a WindowTransition). It is computed as the square root of the sum of the terms' weight squared [93].

ATUA randomly selects related Windows using the dependency score as probability distribution.

Phase 3 terminates when the overall test budget is exhausted or all the instructions of the target methods have been covered.

Running Example. Phase 3 enables ATUA to test the Activity Diary feature that visualize the details of the current activity after a click on the activity name. This feature requires a specific configuration to be enabled in the settings page (by default, a click on the activity name terminates the activity). After selecting the MainActivity as target Window (EditActivity had been fully exercised in Phase 2), ATUA selects the SettingsActivity as related Window to be exercised first (it is reached with Actions A10 and A11 in Figure 3.6). While exercising the SettingsActivity, it deselects the option *Terminate activity by click* (Action A12). After exercising the related Window, ATUA reaches the MainActivity (A13) and triggers the Action that exercises the modified feature (i.e., click on current activity widget - A14). Phase 3 thus enabled ATUA to test the updated feature (i.e., visualize the current activity's details after a click) in a few steps, which is unlikely with random-based, state-of-the-art approaches (see Section 3.4.5 for additional examples).

3.3 ATUA Toolset

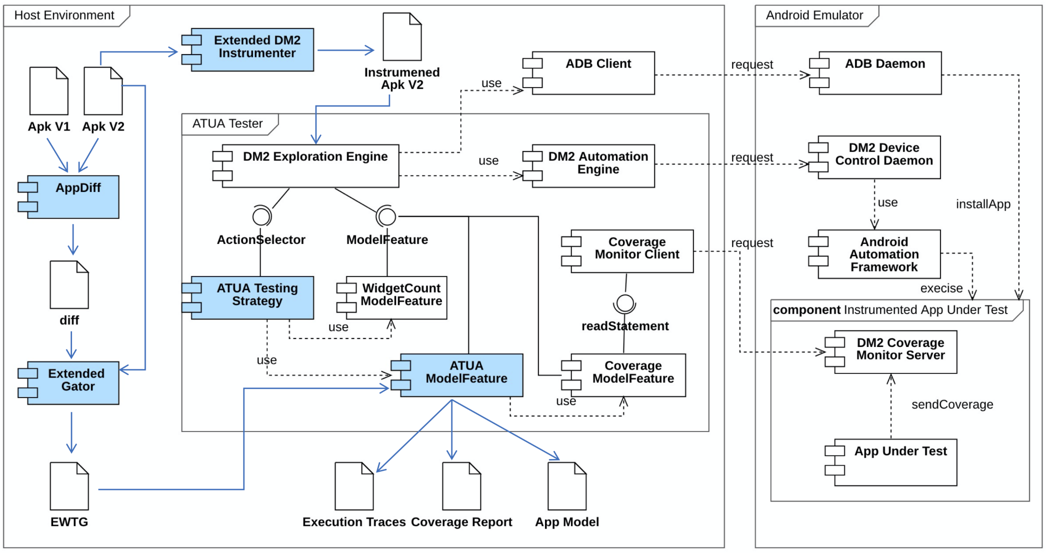
The ATUA Toolset includes four main components: *AppDiff*, which identifies the updated methods for the AUT, *Extended GATOR*, which generates the EWTG part of the AppModel, *Extended DM2 Instrumenter*, which instruments the AUT, and *ATUA Tester*, which implements the ATUA testing algorithm. The UML component diagram in Figure 3.7 shows the ATUA Toolset.

The features of AppDiff and Extended GATOR have already been presented in Sections 3.2.2 and 3.2.3, respectively. In this Section, we focus on the description of the Extended DM2 Instrumenter and the ATUA Tester.

ATUA Tester has been implemented as an extension of DM2. DM2 consists of six components (i.e., DM2 Instrumenter, DM2 Exploration Engine, DM2 Automation Engine, Coverage Monitor Client, Coverage Feature, and Widget Counting Model Feature) that are executed on the host environment and two components (i.e., DM2 Control Device Daemon, and DM2 Coverage Monitor Server) that are deployed on the Android emulator running the AUT. The DM2 components are part of ATUA Tester, which automatically deploys and executes them transparently from the end-user. ATUA Tester integrates two additional components that implement the ATUA algorithm (i.e., ATUA Testing Strategy and ATUA Model Feature). The integration between ATUA Tester components and DM2 is performed through the interfaces provided by DM2 (i.e., ModelFeature and ActionSelector). ATUA Tester and DM2 rely on three additional components provided by the Android development environment: the ADB Client, the ADB Daemon, and the Android Automation Framework.

The *Extended DM2 Instrumenter* is used before testing to create an instrumented version of the AUT that integrates the functions required to collect code coverage. We have extended the *DM2 Instrumenter* to collect method coverage information in addition to instruction coverage. Method coverage is used by ATUA to quickly determine at runtime which methods have been covered.

At runtime, during testing, the *DM2 Exploration Engine* acts as a controller that queries the *ATUA Strategy* component, which implements the DM2 *ActionSelector* interface, used by DM2 to select the next Action to trigger during testing. The *ATUA Strategy* component implements the ATUA's testing algorithm. The DM2 Exploration Engine relies on the *ADB Client* installed on the host to set up the Android emulator and deploy the AUT. The interaction with the AUT is managed by the *DM2 Automation Engine*, which sends commands to the *DM2 Device Control Daemon* installed on the Android emulator. The *DM2 Device Control Daemon* employs the *Android Automation Framework* to execute the requested Action on the AUT and to derive the GUITree for the active Window. After triggering an Action, the *DM2 Automation Engine* receives the current GUITree, a screenshot of the Android emulator GUI, and some additional information (e.g., exception trace from



Legend: White UML component symbols point to third-party, reused components. Blue UML component symbols highlight components developed from scratch or extended to support ATUA's features.

Figure 3.7: Overview of the ATUA toolset.

Logcat) from the *DM2 Device Control Daemon*. It then derives the GUITreeTransition performed on the GSTG and sends this information to all the registered *Model Features*, including the Widget Counting Model Feature, the DM2's Coverage Feature, and the ATUA App Model. The *Widget Counting Model Feature* calculates the frequency of Actions on GUI widgets, used to drive ATUA's random exploration. The *DM2 Coverage Feature* is responsible for tracking code coverage during testing. It associates to each Action the set of instructions covered when executing the Action; code coverage is provided by the *Coverage Monitor Server* instrumented by DM2 within the AUT. The *ATUA Model Feature* updates the App model consistently with the description provided in Section 3.2.1; for example, it implements the ATUA state abstraction mechanism. The *ATUA Model Feature* is queried by the *ATUA Strategy* to determine the Actions to trigger (e.g., to identify the shortest sequence of Actions required to reach a TargetWindow). The *ATUA Model Feature* relies on the *DM2 Coverage Feature* to acquire the coverage data and associate them with the App Model. At the end of the testing, the ATUA Model Feature generates ATUA's outputs (i.e., coverage report, execution traces, and the final App model).

3.4 Empirical Evaluation

The objective of the empirical evaluation is to compare ATUA with state-of-the-art approaches in terms of cost-effectiveness. It is motivated by our need to achieve high test coverage (effectiveness) while enabling the verification of test results within an acceptable budget (cost) in a CI context.

When a new release is ready for testing, a test automation technique is *effective* when it enables engineers to verify updated features in the AUT automatically; more precisely, when it extensively exercises updated methods and their instructions. Measuring the effectiveness of App testing techniques in terms of method and instruction coverage is common practice [13]. Although engineers may aim to exercise all the methods that could be impacted by the changes (e.g., the ones identified by means of change impact analysis as mentioned in Section 3.2), in our empirical evaluation, we focus on updated methods since exercising them is the minimum requirement of any testing criterion targeting software updates.

What we refer to as *cost* comes in two forms, (1) test execution time and (2) human effort, which we define as the time spent by a trained engineer to execute the manual tasks required by a test automation technique. In general, *test execution time* does not necessarily need to be minimized but it should be practical. For example, we expect a test budget of one hour to be practical in a continuous integration context, while a budget of five hours might be acceptable when testing overnight.

Human effort, in our context, is mostly driven from the absence of a solution to automate test oracles, even in the presence of automated test input generation. More precisely, it is not possible to define automated oracles working with any feasible test input; consequently, the evaluation of the correctness of App outputs should, in general, be done manually. As described in Section 3.2, to address the oracle problem in the context of App updates, engineers can rely on two complementary state-of-the-art approaches that respectively address regression failures and failures in newly implemented, repaired, or modified features. To discover regression failures, engineers can replicate, on a previous App version, the test input sequences generated for the updated App. With ATUA, input sequences correspond to the sequences of Actions generated by ATUA to test an App. In such context they address the oracle definition problem by verifying that the outputs generated by the two App versions match. To discover failures in new, repaired, and modified features engineers should visualize the screenshots of the Windows or the GUITrees generated for the provided inputs. The effort in doing so can potentially be reduced through crowdsourcing. In this context, the oracle definition problem remains unaddressed; a manual oracle can be defined (i.e., a human can decide if an output is correct based on the App specifications) but oracle evaluation remains manual and, therefore, expensive. Regardless of the situation and context, human effort, given the scarcity of qualified human resources, should in general be minimized.

We assume that human effort is proportional to the number of inputs generated by test automation. Indeed, to overcome execution errors due to changes in the GUI and be able to replicate test input sequences in a different App version, engineers may need to manually repair the sequence of inputs (e.g., by changing the ID of a widget to be clicked). A large number of inputs may thus lead to numerous repair operations and make test oracle automation infeasible[5]. Also, the number of outputs that shall be visually inspected by an engineer is proportional to the number of inputs; indeed, an engineer shall visualize the GUITree or the screenshots of the active Window rendered after each Action (see Section 3.2). The effort required to inspect a Window or a GUITree depends on the specific output generated by the App and the specification of the App. For example, in our running example (Figure 3.6), it is reasonable to believe that it is simpler to inspect the Window rendered after Action A4, which contains only a text box and a color selector, rather than inspecting the output of Action A10, which leads to a menu dialog with many textual items. However, the specifications of the App may also impact the required human effort. For example, the color visualized when editing an activity (i.e., A4) depends on the color previously selected for the same activity, which requires the engineer to verify such consistency based on execution history. The menu dialog visualized with Action 10, instead, does not depend on execution history and thus may require less human effort for verification. For these reasons, the cost for the visual inspection of results can be estimated only through an experiment involving human subjects. However, since our objective is to *compare the human effort required by different test generation techniques*, not to estimate the cost of failure detection approaches, we can rely on the number of inputs. Indeed, under the assumption that the distribution of the effort for visually inspecting an output is similar for different test automation techniques, we can conclude that techniques leading to a larger number of inputs may lead to proportionally

[5] Related work reports that repairing a single test input takes 15 minutes, on average [94].

increased costs for output verification. For our experiments, such assumption holds because all the considered approaches exercise a common subset of target methods and we expect a same method to lead to similar outputs across different executions. Indeed, we have observed that more than 50% of the target methods exercised in our experiments are covered by all the testing techniques and more than 80% of them are covered by at least two techniques (see Section 3.4.5).

Our research questions are organised according to the two cost measures above. They evaluate the extent to which we have achieved the objectives mentioned in the introduction; RQ1 addresses O2, while RQ2 addresses O1. Further, beyond comparisons, RQ3 aims to characterize how ATUA and state-of-the-art approaches complement each other.

RQ1 *Can ATUA reduce the human effort required for testing Apps, compared to state-of-the-art approaches?* We aim to determine if the number of inputs generated by ATUA is significantly lower than the number of inputs generated by state-of-the-art approaches, for a same execution time budget. A lower number of inputs makes test automation more widely applicable in practice since it reduces the effort related to the definition of test oracles and repair of input sequences. Also, to determine if the effort of using ATUA is justified by practical benefits, we aim to verify if ATUA provides higher effectiveness per unit of effort than state-of-the-art approaches.

RQ2 *Can ATUA effectively test Apps within practical time budgets, compared to state-of-the-art approaches?* This research question aims to determine if ATUA performs significantly better than state-of-the-art approaches in terms of coverage of updated methods and their instructions, for a same execution time budget.

RQ3 *Is there any difference in the functionalities that are automatically exercised across test automation approaches?* We aim to determine if the testing approaches considered in our empirical evaluation are complementary and to which extent. Specifically, we aim to determine if there are differences in the inputs triggered by the different approaches (e.g., input sequence length, widgets being exercised, or program states being reached) that lead to a diverse and complementary set of functionalities being exercised.

A replicability package is made available online [91].

3.4.1 Study subjects

To perform our experiments, we considered as experimental subjects a number of Apps available on the Android Play Store that are highly popular (i.e., more than 100,000 downloads, on average) and that were used for validation in recent papers reporting on related techniques, i.e., APE [37] and DM2 [43]. We considered only the Apps that can be executed on the recent Android simulator version supported by our toolset (i.e., Android API Level above 23). For each App, we considered the latest 10 released versions at the time of running our experiments (hereafter referred to as V0, ..., V9), when available[6]. In Table 4.1, for each version of each subject, we report the overall number of methods, the number of updated methods, the overall number of bytecode instructions, and the number of bytecode instructions belonging to the updated methods. In total, we downloaded and processed 83 App versions, 74 being updated versions.

For all subjects, we treat version V0 as the first released version. The number of updated methods ranges from one (e.g., version V8 of subject App 2) to 1430 (e.g., version V6 of subject App 1), thus being representative of a wide variety of release scenarios (i.e., from simple bug fixes to major releases). The number of bytecode instructions, ranging from 3667 to 165747, shows that the considered Apps have varied degrees of complexity. Further, the growing number of instructions belonging to the different versions of the Apps (e.g., from 32208 to 69856 in the case of Wikipedia) suggests that they are representative of typical Apps where features are incrementally introduced at every release, thus further motivating the adoption of ATUA.

3.4.2 Experimental setup

In our experiments, we compare ATUA with three state-of-the art test automation tools, i.e., DM2, Monkey, and APE. These tools do not specifically target updated methods, but simply aim to maximize the coverage of the whole App.

DM2 has been selected because it is the framework on top of which we implemented ATUA. We configured it to use the biased-random testing strategy, which matches the random input selection strategy of ATUA. The comparison with DM2 enables us to determine if

[6] Preliminary experiments to setup ATUA had been conducted with Jamendo, a music streaming App [95].

Table 3.5: Overview of subject systems.

Subject App		**Details: (V0)**		**V1**	**V2**	**V3**	**V4**	**V5**	**V6**	**V7**	**V8**	**V9**	**V10**
1	Wikipedia	**V**	110	144	146	159	190	198	10239	10263	10264	10269	
		AM	3767	5009	5646	6435	6943	7477	8814	8751	8759	8793	
		UM	3767	446	195	108	370	292	1430	535	13	94	
		AI	32208	38913	43753	48761	51147	54759	68207	69471	69533	69856	
		UI	32208	11000	4157	2441	6606	6345	24536	12724	281	2698	
2	Activity Diary	**V**	105	111	115	117*	118	122	125	130	131	134	
		AM	260	333	333	333	333	450	479	540	540	659	
		UM	260	18	3	7	12	117	39	28	1	49	
		AI	3667	4832	4831	4834	4880	6613	7052	8247	8251	10622	
		UI	3667	558	21	295	599	3393	2027	1535	15	2459	
3	File Manager	**V**	44	53	77	79	82	84					
		AM	2042	2132	3422	3430	3648	3648					
		UM	2042	306	415	11	644	2					
		AI	34389	34931	48241	48294	51755	51789					
		UI	34389	14510	13744	703	24960	143					
4	Nuzzel	**V**	302	303	318*	323	325	328	329	330	331	333	334
		AM	4223	4220	4524	4498	4527	4650	4771	4832	4833	4833	4834
		UM	4223	8	717	75	33	41	21	21	1	1	1
		AI	40522	40449	43309	43083	43403	44234	45331	45908	45913	45916	45940
		UI	40522	151	18335	2952	1593	1990	647	1378	35	69	45
5	Yahoo weather	**V**	1.16.0	1.16.1	1.16.2	1.17.3	1.18.1	1.19.1	1.20.1	1.20.3	1.20.5	1.20.7	
		AM	2932	2904	2904	2630	3105	3109	3178	3255	3255	3303	
		UM	2932	5	4	243	10	16	118	101	12	9	
		AI	38015	37867	37857	34220	38219	38211	39086	39439	39439	39462	
		UI	38015	417	272	10198	857	588	4295	3842	689	961	
6	Wikihow	**V**	2.7.3	2.8.0	2.8.1	2.8.3	2.9.1	2.9.2	2.9.3				
		AM	333	333	333	333	325	322	319				
		UM	333	111	1	1	65	4	18				
		AI	3704	3992	3941	3944	3808	3761	3657				
		UI	3704	2279	39	42	1370	93	543				
7	BBC Mobile	**V**	5.1.0	5.10.0	5.11.0	5.12.0	5.13.0	5.4.0	5.5.0	5.6.0	5.8.1	5.9.0	
		AM	10706	8724	8792	8902	8926	9945	10380	10696	10200	8939	
		UM	10706	649	27	44	25	603	242	553	77	95	
		AI	76649	61604	62078	62937	63232	71053	73082	73950	72439	61618	
		UI	76649	11182	1557	2288	2101	10637	6324	9484	1638	3274	
8	VLC player	**V**	3.1.4	3.1.5	3.1.7	3.2.12	3.2.2	3.2.3	3.2.6	3.2.7	3.2.9		
		AM	6796	6843	6854	8681	8544	8551	8621	8641	8676		
		UM	6796	672	26	3	149	13	51	33	42		
		AI	86266	87560	87886	117207	115344	115412	116409	116647	117071		
		UI	86266	34086	1522	150	9611	1163	3527	1961	3010		
9	City-mapper	**V**	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9	10.0	
		AM	9629	9499	9599	9491	9602	9761	9868	9929	9884	10050	
		UM	9629	51	37	55	73	119	76	73	12	69	
		AI	155117	154086	157036	153200	155950	161914	164267	165747	165747	163303	
		UI	155117	2726	2286	2075	3160	6262	2756	2340	1372	3775	

Legend: V: ID of the version under test. AM: number of methods implemented in V (All Methods). UM: number of Updated Methods in V. AI: number of instructions in V (All Instructions). UI: number of instructions belonging to updated methods. An asterisk (*) is used to indicate not tested versions.

the additional analyses performed by ATUA (i.e., static program analysis, adaptable state abstraction function, and inputs generation based on information retrieval) contribute to generating better results than a simpler solution based on dynamic program analysis only.

Monkey is a program that runs on the Android emulator and generates pseudo-random streams of events. It is used as baseline for App testing approaches and surprisingly fares

Table 3.6: Case studies with manual inputs.

Case study	Feature tested	# Windows	# Data fields	# Instances
Wikipedia	Log-in functionality	1	2	1
	Creation of a new account	1	4	14
VLC	Play a video stream using a URL	1	1	2
	Populate the library with all the videos on the device	1	1	1
Nuzzel	Request an e-mail newsletter	1	1	1

better in many benchmarks [14]. The reason is that the time saved by not processing the App GUITree can be used to further explore the App state space.

APE is a state-of-the-art App testing toolset that overcomes existing approaches thanks to an adaptable state abstraction function (see Section 2).

In our experiments, we considered two possible execution scenarios, with respectively test budgets of one hour (a practical choice in a continuous integration context) and five hours (a reasonable choice for overnight execution).

To use ATUA, for three subjects, we specified a set of manual inputs necessary to exercise the primary features of the Apps (e.g., to login and use the App). Support for manual inputs is a necessary feature of test automation tools because Apps often require domain-specific information that cannot be derived automatically (e.g., login data). Table 3.6 provides a summary of the manual inputs defined; for each, we provide a description, and the number of windows in which the manual input might be triggered. For Wikipedia, we configured two manual inputs, one with the information for creating a new account, another one with information to log-in. In the case of VLC, we provide the URL of a stream to be reproduced and the indication of a checkbox to be checked in order to populate the library with device data (otherwise, no content can be played and testing is limited). Regarding Nuzzel, we provide the e-mail address to receive a newsletter. The effort required to define manual inputs is limited; indeed, for each input, we have specified a single Window where it is applied, between one and four data fields, and a very limited number of input instances, that is, one for every tested feature except for the creation of a new Wikipedia account and the playback of a video stream with VLC. When creating a new account, it is necessary to specify a larger set of inputs to exercise the feature under test multiple times; indeed, a same e-mail address cannot be shared by distinct Wikipedia accounts. As for VLC, since one of its main features is to play video streams, it makes sense to test it with both a working and a corrupted video stream. This example illustrates that the effort required to specify manual inputs is negligible.

To account for randomness, we executed each tool against each updated version 10 times. We report results for 72 of the 74 versions available since, for two App versions of Nuzzel and Activity Diary (indicated with an asterisk in Table 4.1), it was not possible to execute all the testing tools. More precisely, for version 318 of Nuzzel, the App starts but gets stuck in the first Activity, while version 117 of ActivityDiary can be tested only with ATUA and DM2, but not with Monkey and APE. In total, we executed 5760 test sessions (4 tools $\times$ 72 versions $\times$ 10 runs $\times$ 2 test budgets) for a total of 17280 test execution hours. To perform our experiments, we relied on the Grid 5000 infrastructure [96, 97], which provides access to 800 compute-nodes grouped into homogeneous clusters. We rely on nodes with 16x2.1 GHz and 18x2.2 GHz CPU cores.

In the following sections, we analyze differences in results using a non-parametric Mann Whitney U-test (with α = 0.05). Particularly, we discuss the p-values computed by the Mann Whitney U-test to reject null hypotheses stating that there is no difference between ATUA and each of the state-of-the-art solutions, a common practice in software testing research [98, 99]. We discuss effect size based on Vargha and Delaney's A_{12} statistics [100], a non-parametric effect size measure. The A_{12} statistic, given observations (e.g., code coverage, in our context) obtained with two treatments X and Y (testing tools, in our context), indicates the probability that treatment X leads to higher values than treatment Y. Based on A_{12}, effect size is considered small when $0.56 \leq A_{12} < 0.64$, medium when $0.64 \leq A_{12} < 0.71$, large when $A_{12} \geq 0.71$. Otherwise the two populations are considered equivalent [100]. In contrast, when A_{12} is below 0.50, it is more likely that treatment X leads to lower values than treatment Y. Symmetrically to the case above, effect size is small when $0.36 < A_{12} \leq 0.44$, medium when $0.29 < A_{12} \leq 0.36$, large when $A_{12} \leq 0.29$.

3.4.3 RQ1: Human Effort

Experimental setup

In line with the discussion concerning human effort reported above, to address RQ1, we count the number of inputs generated by each testing tool, for each test execution run. For DM2 and ATUA, we rely on the CSV file generated by the ActionTrace component of DM2, which reports all the inputs triggered during testing. For Monkey and APE, we record the number of test inputs reported by the tool at the end of execution.

Metrics For each subject App, we compare *distributions of the number of inputs generated across tools*. We also analyze the *target instructions/input ratio*, that is, the ratio between the number of target instructions (i.e., instructions belonging to updated methods) that are automatically exercised and the number of inputs triggered by the test automation tool. This ratio captures how useful it is for a software engineer, on average, to invest time in repairing a single input of the test sequence or verifying the output produced by an input. For example, a target instructions/input ratio of five indicates that, for every input, the test automation approach exercises, on average, five instructions belonging to updated methods.

To answer positively this research question, ATUA, compared to other tools, should generate fewer test inputs and have the highest *target instructions/input ratio*.

Results

Figures 3.8 and 3.9 show boxplots capturing the number of inputs generated by each approach in every run, for every subject App, for the two distinct test budgets considered (i.e., one hour and five hours). Please note that, in all the boxplots presented in this section: (1) horizontal dashed lines show the average across the data points of the boxplot (i.e., average for the subject App), (2) horizontal dotted lines traversing the whole chart show the average across all the runs, (3) whiskers are used to report min and max values across runs for all versions.

Figures 3.8 and 3.9 shows that ATUA generates, on average, the lowest number of inputs: 876.53 for one hour, 4608.57 for five hours. ATUA is thus the most suitable approach to minimize test automation effort. Monkey generates the largest number of inputs (i.e., 57888.79 for one hour, 291614.69 for five hours) because it does not invest any of the time budget into analyzing execution data but simply generates purely random inputs. APE relies on Monkey to generate inputs; however, APE generates fewer inputs than Monkey (i.e., 27505.89 for one hour, 134640.71 for five hours) because it spends time refining the state abstraction function (see Section 2). Finally, DM2 generates a number of inputs (i.e., 1325.71 for one hour, 6858.16 for five hours) that is closer to those generated by ATUA. This is mostly due to the fact that both approaches are model-based and share the same dynamic analysis infrastructure; however, on average, ATUA generates fewer inputs because it invests more of the time budget into the analysis of run-time data.

Figures 3.10 and 3.11 present the same boxplots as Figures 3.8 and 3.9 but zoom in on ATUA and DM2 data to highlight their differences. Table 3.7 reports the p-value and A_{12} statistics obtained with the Mann Whitney U-test and the Vargha and Delaney's method, respectively. Recall that we aim to minimize the number of inputs and are thus interested in effect sizes below 0.46, i.e., the probability that ATUA generates a number of inputs higher than another approach should be below 0.46, ideally close to zero.

For a budget of one hour, for all the subject Apps, ATUA generates, on average, fewer inputs than DM2. Differences are statistically significant (i.e., we reject the null hypothesis that *there is no difference in the number of inputs generated by ATUA and the state-of-the-art approach*) and effect size is always in favor of ATUA and large for seven of the nine subjects. Differences with Monkey and APE are always significant and effect size is always large except for subject 5 (YahooWeather), in which APE does not interact properly with one App version, apparently because of a bug in APE.

With a budget of five hours, ATUA also generates, on average, fewer inputs than DM2 across subjects. But in the case of Wikipedia, effect size is not in favor of ATUA (i.e., it is likely to generate more inputs than DM2). However, this is due to a bug in DM2 rather than a feature; indeed, in the presence of WebViews, the communication between the DM2 device daemon and the DM2 client are delayed, thus reducing the number of inputs being generated. In the case of ATUA, which is built on top of DM2, this problem is less evident because such communication is triggered less frequently. The differences between the number of inputs generated by ATUA and the ones generated by APE and Monkey are always statistically significant with a large effect size in favor of ATUA (except for YahooWeather, as already discussed above).

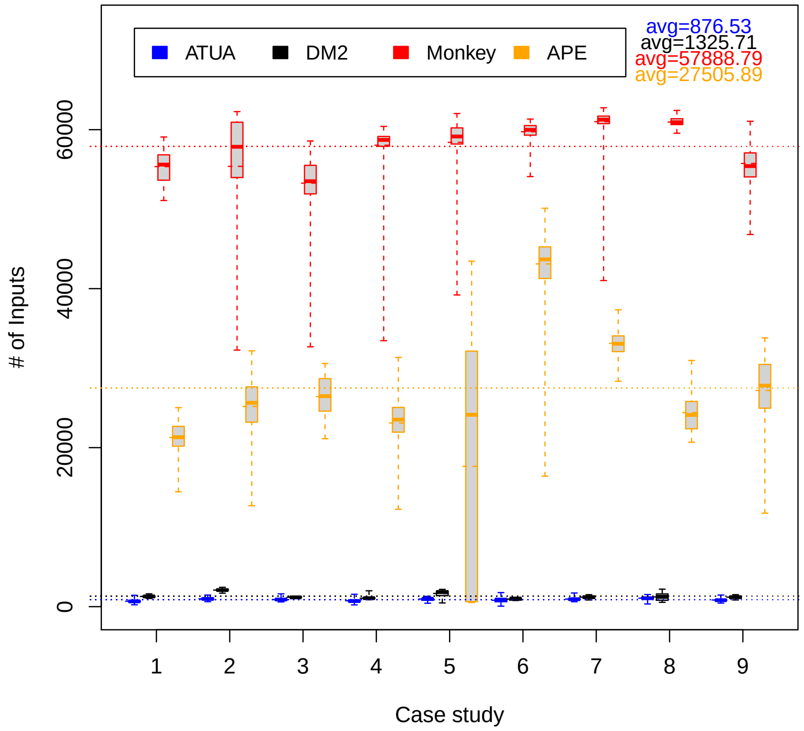


Figure 3.8: Number of inputs generated (budget=1 hour).

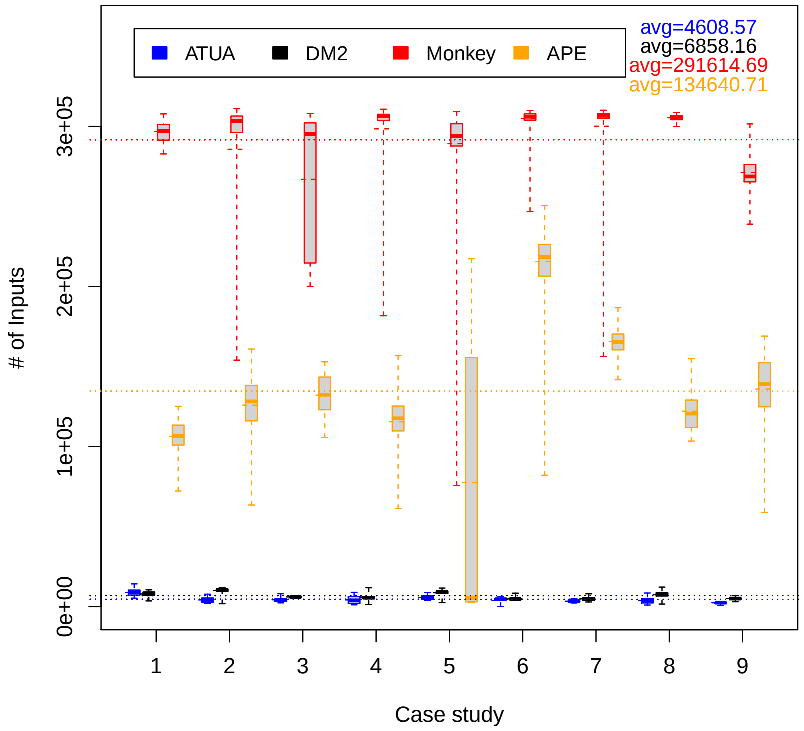


Figure 3.9: Number of inputs generated (budget=5 hours).

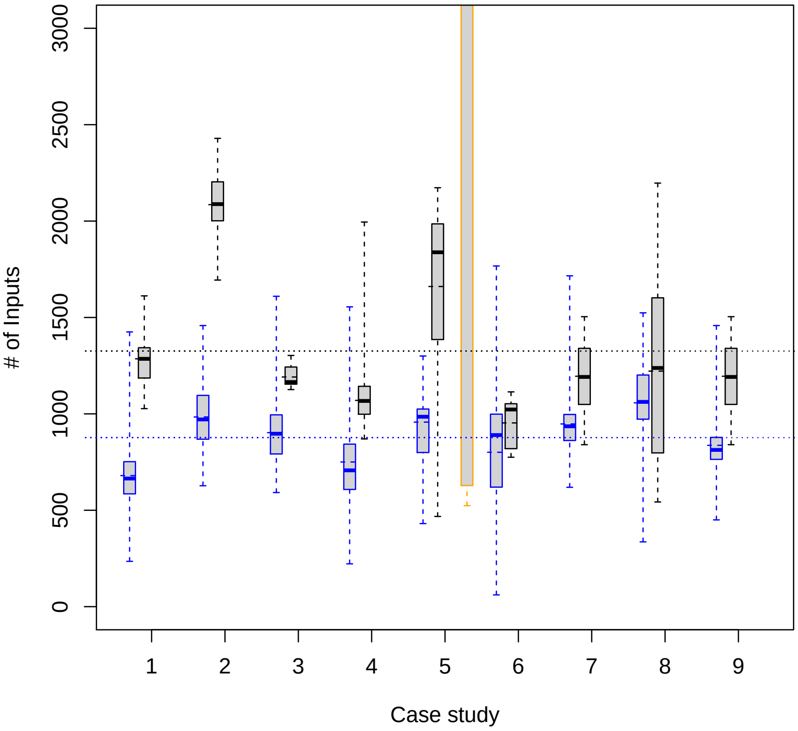


Figure 3.10: Number of inputs generated (budget=1 hour). Zoom on ATUA and DM2 data.

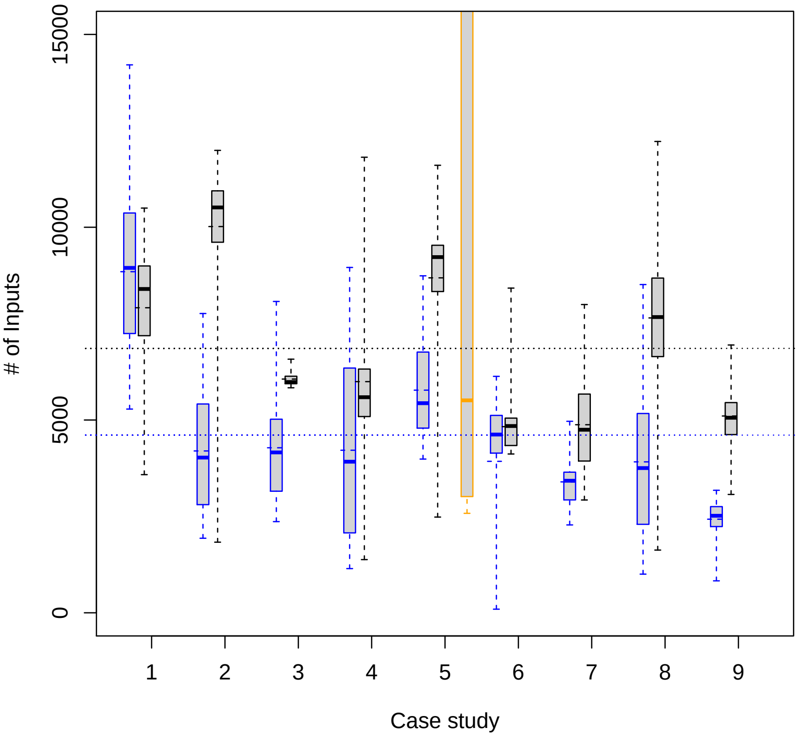


Figure 3.11: Number of inputs generated (budget=5 hours). Zoom on ATUA and DM2 data.

Table 3.7: Statistical significance and effect size for Figure 3.8 and 3.9.

	1 hour budget						**5 hours budget**					
	p-value			**A_{12}**			**p-value**			**A_{12}**		
C	D	M	A	D	M	A	D	M	A	D	M	A
1	<0.05	<0.05	<0.05	.018	.000	.000	<0.05	<0.05	<0.05	.630	.000	.000
2	<0.05	<0.05	<0.05	.000	.000	.000	<0.05	<0.05	<0.05	.020	.000	.000
3	<0.05	<0.05	<0.05	.039	.000	.000	<0.05	<0.05	<0.05	.093	.000	.000
4	<0.05	<0.05	<0.05	.099	.000	.000	<0.05	<0.05	<0.05	.278	.000	.000
5	<0.05	<0.05	<0.05	.074	.000	.397	<0.05	<0.05	0.81	.090	.000	.490
6	<0.05	<0.05	<0.05	.360	.000	.000	0.15	<0.05	<0.05	.425	.000	.000
7	<0.05	<0.05	<0.05	.117	.000	.000	<0.05	<0.05	<0.05	.123	.000	.000
8	<0.05	<0.05	<0.05	.405	.000	.000	<0.05	<0.05	<0.05	.075	.000	.000
9	<0.05	<0.05	<0.05	.048	.000	.000	<0.05	<0.05	<0.05	.001	.000	.000

Legend: *S*, subject. *D*, comparison with DM2, *M*, Monkey, *A*, APE. We underline the few cases in which statistics indicate that ATUA shows no significant difference (i.e., *p-value* ≥ 0.05) or no higher chances of generating less instructions ($A_{12} > 0.44$) than state-of-the-art approaches.

Figures 3.12 and 3.13 show, for each subject, the distribution of the target instructions/inputs ratio. ATUA has the highest ratio: 2.26 for one hour, 0.49 for five hours. For the one-hour budget, ATUA's test automation effort (i.e., manual repair of a GUI input, visual inspection of outputs) is thus more beneficial because each input enables the verification of 2.26 additional target instructions. As a comparison, other state-of-the-art approaches yield lower ratios: 1.34 (DM2), 0.03 (Monkey), and 0.10 (APE). These results show that, though Monkey and APE are known for effectively triggering crashes, they are unlikely to be applicable in a testing context where the number of generated inputs should be minimized. For a time budget of five hours, average differences are less pronounced but the same trends hold.

Table 3.8 provides p-values and A_{12} statistics. Since we aim to determine if ATUA is likely to generate a higher target instructions/inputs ratio, we look for A_{12} values above 0.50. For a one-hour budget, effect size is always in favor of ATUA (i.e., is more likely to generate a higher instructions/inputs ratio); effect size is always large with respect to Monkey and APE. Even if in a few cases differences are not statistically significant (i.e., we cannot reject the null hypothesis that *there is no difference between ATUA and the state-of-the-art approach concerning the ratio between instructions covered and inputs being triggered*), effect size trends provides a clear picture of the benefits: **ATUA is likely to yield a higher instructions/inputs ratio**. The same conclusions can be drawn for a five-hour budget, though for two subjects (Wikipedia and VLC) ATUA performs similarly to DM2.

To summarize, regarding the human effort required for practical execution time budgets, ATUA performs better than the other approaches since it saves around 33.8% (1 hour budget)

Table 3.8: Statistical significance and effect size for target instructions/inputs ratios.

	1 hour budget						**5 hours budget**					
	p-value			$\mathbf{A_{12}}$			**p-value**			$\mathbf{A_{12}}$		
C	D	M	A	D	M	A	D	M	A	D	M	A
1	0.10	<0.05	<0.05	.728	1.00	.975	0.69	<0.05	<0.05	.555	.963	.901
2	0.17	<0.05	<0.05	.703	.906	.875	0.14	<0.05	<0.05	.719	.938	.875
3	0.29	0.09	0.09	.700	.820	.820	0.25	0.08	0.17	.720	.840	.760
4	0.33	<0.05	0.05	.636	.895	.772	0.38	<0.05	<0.05	.623	.895	.747
5	0.17	<0.05	<0.05	.691	1.00	.901	0.17	<0.05	<0.05	.691	.950	.827
6	0.63	<0.05	<0.05	.583	1.00	1.00	0.52	<0.05	<0.05	.611	1.00	1.00
7	0.27	<0.05	<0.05	.654	1.00	1.00	0.27	<0.05	<0.05	.654	1.00	1.00
8	0.60	<0.05	<0.05	.578	1.00	.953	0.92	<0.05	<0.05	.516	.968	.906
9	0.39	<0.05	<0.05	.642	1.00	1.00	<0.05	<0.05	<0.05	.914	1.00	1.00

Legend: *S*, subject. *D*, comparison with DM2, *M*, Monkey, *A*, APE. We underline the few cases in which statistics indicate that ATUA shows no significant difference (i.e., *p-value* ≥ 0.05) or no higher likelihood of achieving a higher instructions/inputs ratio (i.e., $A_{12} < 0.56$) than state-of-the-art approaches.

and 32.8% (5 hours budget) of the effort compared with DM2, while it shows huge savings compared to the other two approaches. As for the effectiveness per unit of effort, ATUA provides tangible gains of 68.7% (1h) and 63.3% (5h) compared with DM2, and huge differences with the others. **ATUA therefore significantly decreases the human effort required for repairing inputs and defining oracles when compared to state-of-the-art approaches.**

3.4.4 RQ2: Effectiveness within Time Budget

Experiment design

To address RQ2, we focus on code coverage results obtained with the updated versions of our subject Apps, i.e., versions V1 to V9. More precisely, we keep trace of the updated methods (hereafter, *target methods*) and instructions belonging to updated methods (hereafter, *target instructions*) that are exercised by the test automation tools considered in our study. To collect data for ATUA and DM2, we rely on the Soot-based code coverage extension integrated into DM2; for Monkey and APE, we rely on MiniTracing, a toolset developed to measure code coverage with APE [101]. Since all these code coverage tools measure the coverage of the whole AUT, to determine the coverage of target methods and instructions, we filter results based on the list of updated methods generated by AppDiff.

Since ATUA and state-of-the-art approaches require a different degree of human effort (see RQ1) and human effort is measured in terms of inputs generated, we shall set an identical limit to the number of inputs that might be generated by the test generation techniques. The

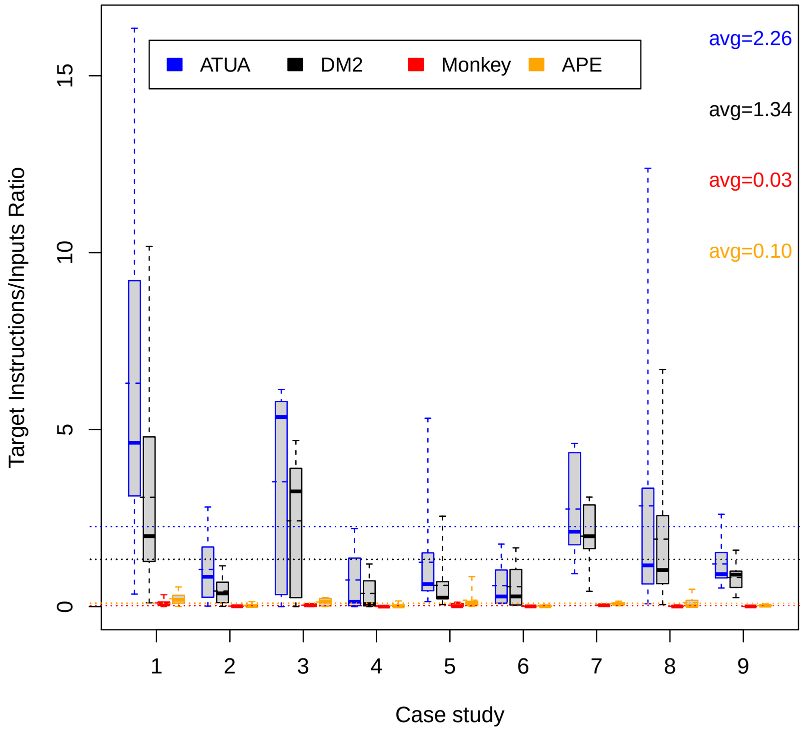


Figure 3.12: Target instructions/inputs ratio (budget=1 hour).

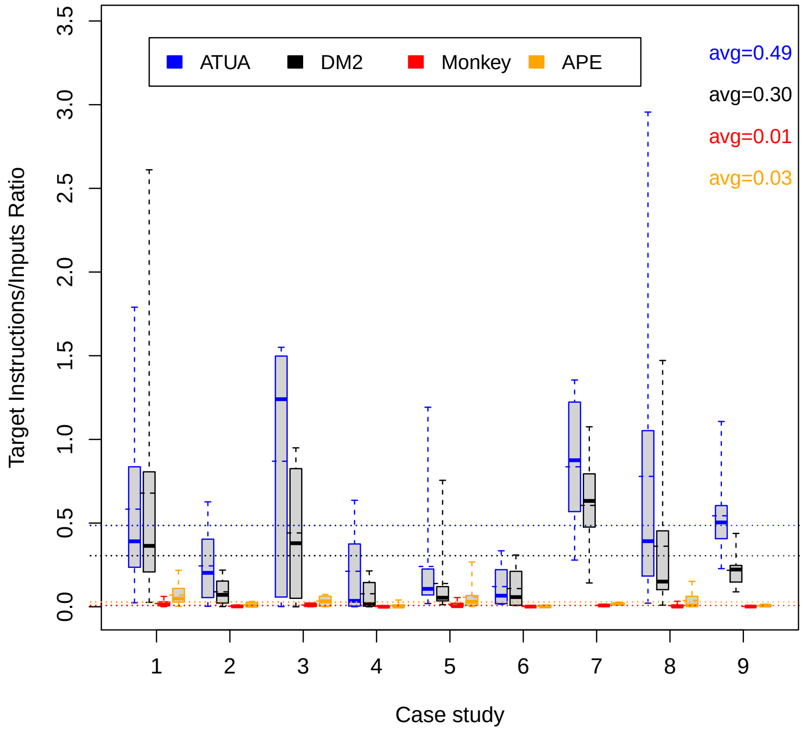


Figure 3.13: Target instructions/inputs ratio (budget=5 hours).

rationale is that we try to emulate, in our experiments, realistic conditions where testers are limited by both execution time and human resources. This is thus expected to yield unbiased comparisons of practical value. It is also consistent with our objective, stated earlier, of minimizing human effort while keeping execution time within acceptable bounds. More precisely, for each software version v, we define an inputs budget equal to the maximum number of inputs generated, over ten runs, by ATUA, which is the approach generating the fewest test inputs for a given time budget, based on RQ1 results.

Though the fault detection rate (i.e., the proportion of faults being detected by a test automation technique) would be a useful, complementary metric to evaluate test automation effectiveness, it is inapplicable in our context since an important subset of our subject Apps (BBC, YahooWeather, Wikihow, Nuzzel) are not open source, a choice made to include representative Apps. Indeed, the unavailability of source code and bug repositories prevented us from determining if a failure was due to a fault introduced by an App upgrade. Further, supporting the use of coverage for our experiments, it has been recently shown that there is moderate to high correlation between code coverage and the detection of real faults [102]. Finally, without automated functional oracles, in existing studies, effectiveness is typically measured by looking at runtime failures (i.e., due to uncaught exceptions or crashes), which represent only a small proportion of the failures that are typically observed in the field [103].

Metrics Since the number of target methods and instructions varies across App versions, ATUA and state-of-the-art approaches shall be compared in terms of percentage of target methods and percentage of target instructions covered. In addition, such coverage metrics shall be obtained when testing a subject App for a maximum and practical execution time budget (i.e., one hour and five hours, as discussed in Section 3.4.2), while not exceeding a maximum input budget determining human effort.

To positively answer this research question, ATUA should, in statistical terms, exercise a larger percentage of target methods and instructions than the other approaches.

Results

Figures 3.14 and 3.16 show the distribution of the percentage of target methods and instructions that have been covered by the selected testing tools for the subject Apps, with a test budget of one hour. Figures 3.15 and 3.17 report the same measurements for a budget of five hours.

With a test execution budget of one hour, ATUA is the approach with the highest percentage of target methods and instructions being exercised on average, with 66.34% and 56.14%, respectively. The largest differences are observed when ATUA is compared to Monkey; indeed, on average, ATUA exercises 33.46% and 27.42% more methods and instructions than Monkey, respectively. Since Monkey implements a pure random exploration strategy, our results show that a limit on the number of inputs generated by Monkey highly affects its performance. In contrast, the APE state abstraction function enables a more effective generation of test inputs, thus leading to, on average, higher coverage than Monkey. However, ATUA outperforms APE; indeed, on average, ATUA exercises 21.66% and 18.41% more target methods and instructions than APE, respectively. Though DM2 fares better than Monkey and APE, as it relies on a model-based approach leveraging dynamic analysis, ATUA exercises 7.50% and 6.37% more target methods and instructions. Note that, the increase achieved by ATUA is particularly significant, +12.74% (i.e., +7.50%/58.84%) and +12.79% (i.e., +6.37%/49.77%) for methods and instructions coverage, respectively. This is explained by the transition-driven exploration based on static analysis (ATUA Phase 1 and 2) and information retrieval (Phase 3), which are not part of DM2.

When executed with a test budget of one hour, for all the subject Apps, both the median and the average obtained with ATUA are higher than those obtained with other approaches.

To discuss differences across subjects, we report in Table 3.9 the p-value and A_{12} statistics obtained with the Mann Whitney U-test and Vargha and Delaney's method, respectively. Overall, differences are statistically significant[7] but there are exceptions: when ATUA is compared to APE for Nuzzel (subject 4), and when ATUA is compared to DM2 for Nuzzel, File Manager (subject 3), Wikihow (subject 6), and VLC (subject 8). However, for most of the subjects, ATUA is likely to exercise more target methods and instructions than other approaches; this is shown by the A_{12} statistics being always above 0.56, except for File Manager, Wikihow, and VLC in the case of DM2[8]. Regarding VLC, the effectiveness of ATUA is limited by the need for setup operations that require some human effort. Indeed,

[7] We can reject the null hypothesis that *there is no difference in the number of target methods and instructions exercised by ATUA and the i-th state-of-the-art approach.*

[8] Average A_{12} is 0.77 for target methods and 0.76 for target instructions coverage; median is 0.80 for target methods and 0.77 for target instructions coverage.

Table 3.9: Statistical significance and effect size for RQ2.

	1 hour budget						**5 hours budget**					
	p-value			**A_{12}**			**p-value**			**A_{12}**		
C	D	M	A	D	M	A	D	M	A	D	M	A
	Coverage of target methods											
1	<0.05	<0.05	<0.05	.857	.994	.956	<0.05	<0.05	0.14	.703	.935	.564
2	<0.05	<0.05	<0.05	.733	.868	.703	<0.05	<0.05	<0.05	.774	.832	.623
3	0.41	<0.05	<0.05	.548	.795	.684	0.61	<0.05	0.99	.529	.767	.501
4	0.16	<0.05	0.07	.560	.852	.577	0.16	<0.05	0.08	.560	.760	.575
5	<0.05	<0.05	<0.05	.735	.814	.920	<0.05	<0.05	<0.05	.713	.702	.841
6	0.08	<0.05	<0.05	.587	.838	.689	0.2	<0.05	<0.05	.564	.719	.681
7	<0.05	<0.05	<0.05	.614	.836	.762	0.78	<0.05	<0.05	.488	.879	.879
8	0.23	<0.05	<0.05	.554	.997	.972	0.05	<0.05	<0.05	.588	.995	.892
9	<0.05	<0.05	<0.05	.587	.981	.878	<0.05	<0.05	<0.05	.685	.992	.918
	Coverage of target instructions											
1	<0.05	<0.05	<0.05	.853	.991	.936	<0.05	<0.05	0.15	.735	.884	.562
2	<0.05	<0.05	<0.05	.707	.843	.783	<0.05	<0.05	<0.05	.756	.829	.721
3	0.31	<0.05	0.14	.559	.774	.585	0.18	<0.05	0.70	.577	.705	.478
4	<0.05	<0.05	0.07	.600	.861	.579	0.05	<0.05	<0.05	.584	.764	.594
5	<0.05	<0.05	<0.05	.684	.748	.886	<0.05	<0.05	<0.05	.659	.623	.822
6	0.42	<0.05	<0.05	.542	.788	.827	0.47	<0.05	<0.05	.538	.727	.786
7	<0.05	<0.05	<0.05	.598	.735	.695	0.65	<0.05	<0.05	.480	.780	.780
8	0.06	<0.05	<0.05	.586	.999	.984	0.33	<0.05	<0.05	.633	.999	.936
9	0.05	<0.05	<0.05	.584	.980	.822	<0.05	<0.05	<0.05	.671	.988	.896

Legend: *C*, case study. *D*, comparison with DM2, *M*, Monkey, *A*, APE. We underline the few cases in which statistics indicate that ATUA shows no significant difference (i.e., *p-value* ≥ 0.05) or no higher likelihood (i.e., $A_{12} < 0.56$) of covering more targets than state-of-the-art approaches.

since certain features can be tested only on specific devices (e.g., an Android TV), identifying target methods through static analysis is of limited usefulness and ATUA performs similarly to DM2. However, such limitations could be surmounted after investing some effort to carefully setup ATUA. For example, by configuring ATUA to be executed on an Android TV in addition to a mobile emulator (i.e., what we used in our experiments). Concerning File Manager and Wikihow, ATUA is affected by some limitations of static analysis, which cannot determine that certain WindowTransitions are associated to specific data types provided as input. More precisely, in the case of File Manager, a number of updated features can be exercised only through specific files (e.g., the decompress operation can be executed only with files having ZIP or RAR filename extension). The static analysis currently implemented in ATUA cannot determine that certain features are enabled only in the presence of specific runtime data (e.g., file names) and thus ATUA, similar to DM2, exercises such features only if it accidentally triggers them thanks to random exploration. A similar but more evident problem occurs also in the case of Wikihow, where static analysis does not identify the WindowTransitions triggered by the inputs sent to WebViews. Indeed, the input handlers executed after sending an input to a WebView (e.g., a click on an anchor) depend on the content of the page (e.g., the file type appearing in the URL of the anchor) and thus cannot be identified by static analysis, which does not process the content of the HTML pages displayed at runtime. For this reason, ATUA cannot fully take advantage of static analysis results in the presence of WebViews. In such cases, similar to DM2, ATUA exercises App features thanks to random exploration. However, current ATUA results with Apps using WebViews largely depend on the proportion of features implemented through WebViews. For example, in the case of WikiHow, which mainly relies on WebViews (five out of eight content types are displayed through a WebView), ATUA performs similarly to DM2; instead, in the case of Wikipedia, which implements only one out of 35 Windows using a WebView[9], ATUA outperforms all the other approaches ($A_{12} \geq 0.56$). To overcome the limitations of static analysis and thus improve ATUA results, it might be necessary to develop dedicated strategies relying on dynamic analysis; for example, by extending the state abstraction function of ATUA to use reducers dedicated to HTML anchors or file objects.

With a test budget of five hours, all the approaches achieve better coverage results; however, the ranking observed for a one-hour budget remains unchanged. ATUA is the approach with the highest percentage of exercised target methods and instructions, on average, with 70.37% and 60.18%, respectively. The largest differences are still observed when ATUA is compared to Monkey; indeed, on average, ATUA exercises 27.24% and 22.73% more target methods and instructions than Monkey, respectively. ATUA exercises 18.38% and 15.77% more target methods and instructions than APE. The second best approach remains DM2, as ATUA exercises 6.45% and 6.25% more target methods and instructions than DM2, with a gain of +10.09% (+6.45%/63.92) and +11.59% (+6.25%/53.93%) in the number of target methods and instructions covered with respect to DM2, respectively.

A larger time budget enables ATUA to achieve higher coverage. This is set with the *scaleFactor* configuration parameter (see Section 3.2.5), which we increase for a five-hour budget, thus augmenting the time spent to perform random exploration, reach the test target, and exercise targets.

With a test budget of five hours, the difference between ATUA and other approaches decreases though. Unsurprisingly, with a larger test budget, random-based approaches can more easily reach updated features than with a one-hour budget, for which leveraging static analysis is more important. For example, in subject App 7 (BBC Mobile), in five hours, DM2

[9] In Wikipedia, WebViews are used to display Wikipedia pages while other Views are used for other features such as displaying news, image galleries, or editing the content of a page.

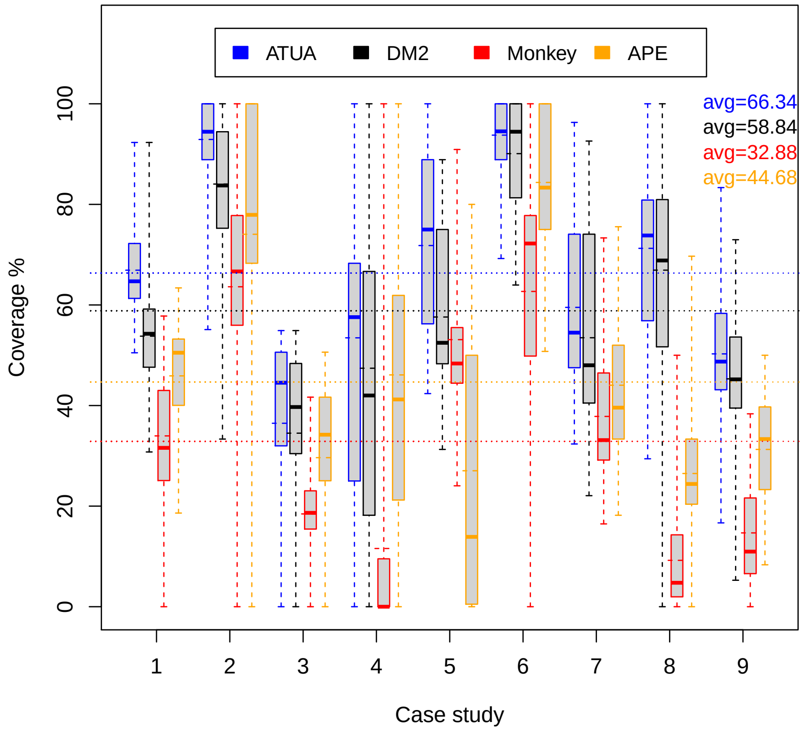


Figure 3.14: Percentage of updated methods covered for each version of the case studies (budget=1 hour).

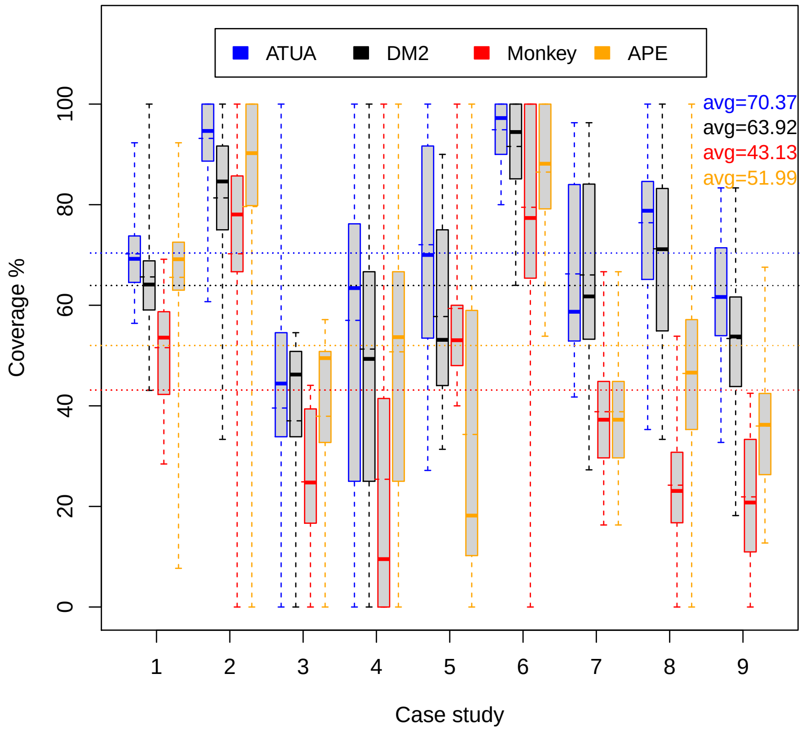


Figure 3.15: Percentage of updated methods covered for each version of the case studies (budget=5 hours).

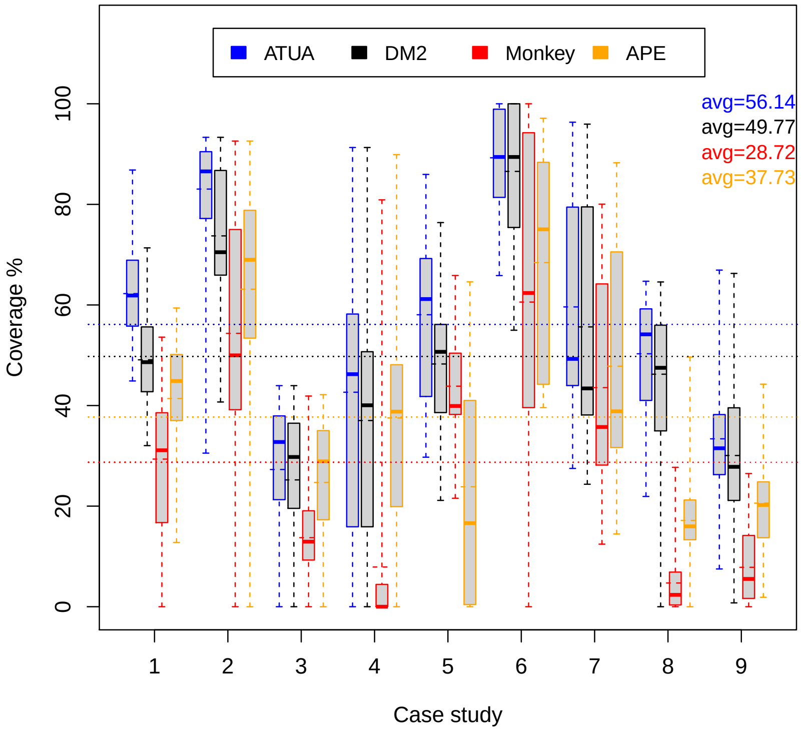


Figure 3.16: Percentage of instructions belonging to updated methods that are covered for each version of the case studies (budget=1 hour).

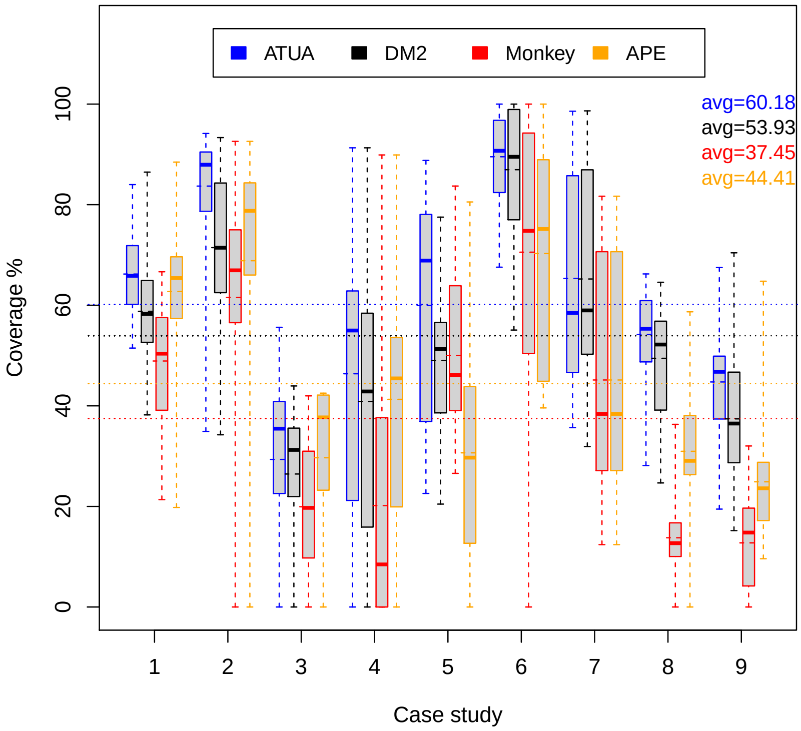


Figure 3.17: Percentage of instructions belonging to updated methods that are covered for each version of the case studies (budget=5 hours).

achieves the same average coverage as ATUA.

To discuss differences across subjects, we refer to the p-value and A_{12} statistics reported in the rightmost columns of Table 3.9. Because of the larger test budget benefiting random exploration, differences between ATUA and other approaches are not significant in 7 out of 27 cases (i.e., 3×9, which is the number of pairwise comparisons between ATUA and the other approaches), two cases more than with a one-hour budget. However, ATUA is still likely to exercise more target methods and instructions than other approaches. Indeed, for both method and instruction coverage, the A_{12} statistics is above 0.56 for 24 out of 27 cases. In general, effect size is slightly lower than for a one-hour budget, with an average A_{12} of 0.73 and 0.72 for the coverage of target methods and instructions. In particular, we observe that the larger time budget enables random-driven approaches to achieve the same effectiveness as ATUA when ATUA is negatively affected by static analysis limitations. This happens for File Manager (subject 3), where APE performs similarly to ATUA, Wikihow (subject 6), where DM2 performs similarly to ATUA, and BBC mobile (subject 7), where the additional time budget enables DM2 to exercise the few updated features depending on WebViews (in BBC Mobile, WebViews are used to display BBC Web pages).

To summarize, ATUA is the approach that, on average, most effectively test updated Apps within practical time budgets and human effort. It tends to cover more target methods and instructions than other approaches. The second best approach is DM2. For a one-hour budget, on average, ATUA automatically exercises 7.50% and 6.37% more target methods and instructions than DM2, with a gain of +12.74% and +12.79%, respectively. With a five-hour budget, on average, ATUA automatically exercises 6.45% and 6.25% more target methods and instructions than DM2, with a gain of +10.09% and +11.59%, respectively. For seven out of nine subjects, for both time budgets, ATUA tends to exercise more target methods and instructions than DM2. For the remaining two subjects, DM2 and ATUA are comparable, due mostly to the current limitations of static analysis.

3.4.5 RQ3 - Complementarity of Testing Approaches

Experiment design.

A software testing approach is complementary to other approaches if it exercises a set of functionalities not exercised by the others. Since we measure effectiveness based on method coverage, to determine complementarity, we look for methods that are univocally covered by each testing approach considered in our experiments. A method is *univocally covered* by approach A for version V of a subject App S if it is exercised by A in at least one of the ten test execution runs on version V and is not exercised by any other approach in any test execution run of that same version. We cannot compare testing approaches based on instruction coverage because some of our subjects are commercial Apps released without source code (e.g., to understand the semantics of the covered instructions). Since the number of target methods varies for each App version, we compare the percentage of target methods that are univocally covered by each approach. Finally, it shall be possible to identify common characteristics in the inputs triggering the univocally exercised methods.

Metrics To compare testing approaches, we thus report (1) the *overall number of univocally covered methods across all the subject App versions* and (2) *the distribution of the percentage of tested methods that are univocally covered by each approach, across all the subject App versions*. Furthermore, we manually inspect the list of univocally covered methods. Based on their signatures[10] and, for ATUA, the data collected in the GSTG, we (3) *determine the characteristics of the inputs and upgraded functionalities that are better targeted by each of the testing approaches*. Since the test budget affects the performance of testing approaches, we discuss the results achieved for one-hour and five-hour budgets, separately.

For each testing approach, we analyze if (1) it covers a large number of methods not covered by other approaches across all Apps, (2) its distribution, across Apps, of the percentage of tested methods not covered by other approaches has a significantly larger average than that of other approaches, and (3) we can characterize the situations in which the approach univocally exercises some updated App methods.

[10] Since most of our subject Apps are commercial Apps released without source code, the functionality implemented by a method is inferred from its signature.

Results.

For the one-hour budget, a total of 6982 methods belonging to the different App versions have been exercised in our experiments. Overall, 784 out of 6982 methods (11%) are exercised only by one testing approach, 6198 methods (89%) are covered by at least two approaches, while 3648 methods (52%) are covered by all the approaches. ATUA exercises the largest number of methods not exercised by other approaches, 518 (66%); it is followed by APE (156, i.e., 20%), DM2 (78, i.e., 10%), and Monkey (32, i.e., 4%).

Figure 3.18 shows the distribution of the percentage of methods exercised in our experiments that are univocally covered by one testing approach, across versions, for a one-hour budget. On average, 7% of the methods exercised in our experiments are covered only by ATUA, while the other approaches univocally cover only 1% or less. Differences are statistically significant. Also, we report that for 52% of the App versions, ATUA exercises more univocally covered methods than other approaches (92% if including versions with the same number of univocally covered methods). These results show that, across a majority of individual versions, ATUA provides coverage capabilities that cannot be obtained with other approaches.

The effectiveness of ATUA is primarily due to its capability of reaching target Windows and target Widgets that are difficult to reach by solely relying on random exploration. We identify three distinct cases. *First*, ATUA can trigger complex sequences of inputs that enable the visualization of target Widgets. This is the case of SettingsActivity for Nuzzel, which requires opening a drawer, swipe up, and then click on the settings button. Similarly, in Yahoo Weather, it is necessary to swipe up the weather information fragment and click on the map to trigger methods on the WeatherMapView. Other similar cases concern the renaming of files and the opening of the preferences activity in File Manager. Such complex sequences of events are unlikely to be triggered by approaches relying on random exploration; instead, they are selected by ATUA thanks to the use of the App model to identify both the sequence of events that reaches a target Window and the target events that exercise the updated methods. The *second* case concerns ATUA being able to bring an App into a specific state required for testing, which is enabled by the fact that, in Phase 2, ATUA exercises the inputs that trigger updated methods multiple times, when the App has likely reached different App states. For example, ATUA is the only approach exercising the method *undeleteActivity* of

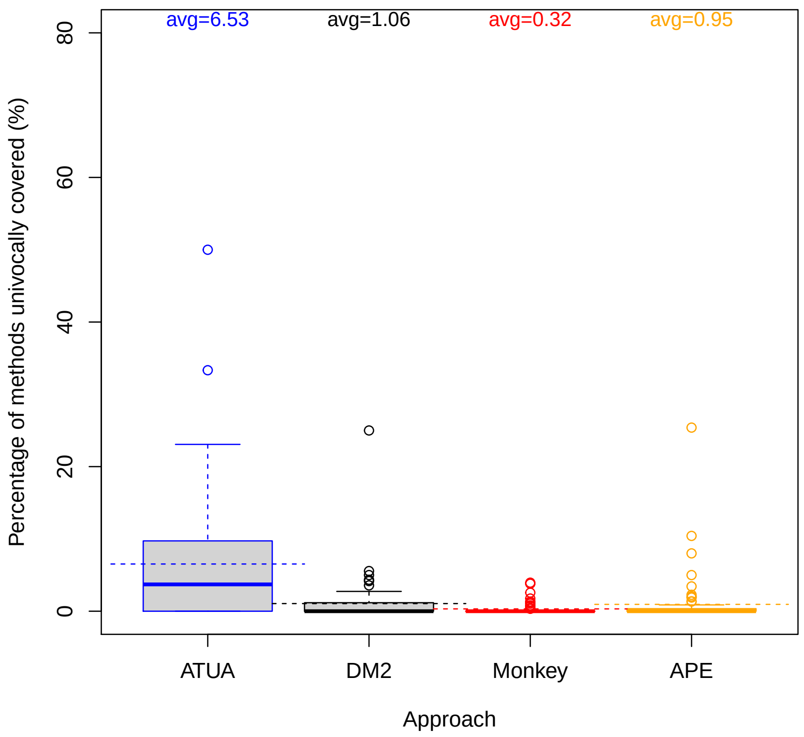


Figure 3.18: Proportion of methods that are univocally covered by one testing approach, distribution across all the tested subject versions (budget=1 hour).

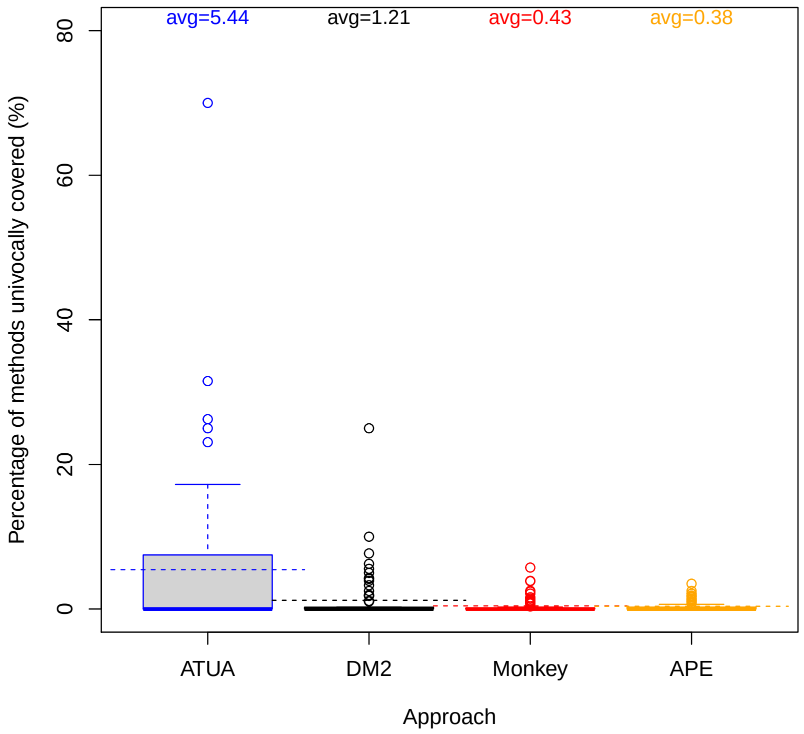


Figure 3.19: Proportion of methods that are univocally covered by one testing approach, distribution across all the tested subject versions (budget=5 hours).

class *ActivityHelper* in Activity diary, which requires to first create an activity, then delete it, and finally undelete it. The *third* case concerns ATUA's capacity to select the Apps' options required to exercise certain Apps' features, which is the objective of Phase 3. For example, in BBC mobile, to exercise the methods in class *MyNewsByTimeFragment*, it is necessary to reach the settings window, enable the option *My News By Topic*, and then open the tab *My News*.

The methods not covered by ATUA but covered by other approaches, instead, are generally the ones whose triggering Actions cannot be identified by ATUA because of the limitations of static analysis. We have identified three different scenarios in which ATUA is less effective than state-of-the-art approaches. First, though ATUA can trigger complex input sequences, it cannot, in certain cases (e.g., in classes extending SettingsActivity), identify the events that trigger specific WindowTransitions, which APE can instead trigger. By relying on an adaptable state abstraction function, APE can direct random exploration towards App states that contain the widgets required to test the updated methods, which Monkey and DM2 do not achieve. For example, this happens when testing the updated methods of *AboutActivity* in File Manager. Second, in the case of WebViews, instead, DM2, by investing more budget on random exploration, can reach Windows that can't be reached by ATUA because static analysis does not identify the required WindowTransitions in the EWTG. Also, based on the observed results, the specific random exploration strategy implemented by DM2 appears to be more effective than the one of APE and Monkey. Finally, Monkey performs better than ATUA and the other approaches when the execution of updated methods depends on specific environmental conditions; for example, the internet connection being disabled, which is the case for testing methods *onGoOffline* and *onPageLoadError* in Wikipedia.

Similar findings can be observed for a test budget of five hours. Overall, a total of 7326 methods have been exercised in our experiments, which is expectedly higher than for the one-hour budget. Overall, 675 out of 7326 methods (9%) are exercised only by one testing approach, 6767 methods (92%) are covered by at least two approaches, while 4308 methods (58%) are covered by all approaches. Although the larger test budget enables all the approaches to exercise a larger common set of methods, we still observe a high degree of complementarity (i.e., 9% of the covered methods are univocally covered). In particular, ATUA remains the approach that exercises the largest number of methods not exercised by other approaches: 478 (71%); the other approaches, instead, show similar numbers of univocally covered methods: 71 for APE, 64 for DM2, 62 for Monkey.

Figure 3.19 depicts the distribution of the percentage of covered methods that are univocally covered by one testing approach, across versions, for a budget of five hours. On average, across versions, 5% of the methods exercised in our experiments are covered only by ATUA, while the other approaches univocally cover only 1% or less, thus confirming that, even for a test budget of five hours, ATUA complements all the other approaches. Differences are statistically significant. Such complementarity is also stressed by the fact that for 42% of the App versions, ATUA covers more univocally covered methods than other approaches (81% if including versions with the same number of univocally covered methods).

Also, for a five-hour budget, ATUA confirms its capacity to trigger complex sequences of inputs not generated by other approaches. This is the case for File Manager, where ATUA successfully starts the FTP client by (1) clicking on the "add" button, (2) then clicking on "Cloud connection", (3) then clicking on SCP/SFTP connection and (4) finally, within the SCP/SFTP connection dialog, fill all the compulsory fields and (5) click on the "Create" button. Such a complex sequence of inputs (including filling FTP connection information) is unlikely to be generated by random approaches. ATUA, instead, once it finds the sequence of inputs that reaches the SCP/SFTP connection dialog, can, in Phase 2, trigger multiple sequences of inputs until it (randomly) finds the one that successfully starts the FTP client. For APE, Monkey, and DM2, we can observe the same characteristics observed as for a one-hour test budget.

To summarize, for both one-hour and five-hour budgets, **ATUA is the approach that exercises the largest number of univocally covered methods**. Across versions, the percentage of exercised methods univocally covered by ATUA is significantly larger than that of other approaches. In practice, the results above also suggest that it might be useful to combine approaches since they may complement each other to cover a larger number of methods. However, ATUA should always be included in the selected combination since it exercises a larger set of upgraded functionalities that cannot be exercised using other approaches.

3.4.6 Discussion

Human effort RQ1 results have shown that ATUA performs better than the other approaches since it saves around 33.8% (one-hour budget) and 32.8% (five-hour budget) of the effort compared with DM2, the second-best approach. Hereafter, we discuss practical implications concerning testing costs based on related work about the nature of App upgrades [3] and the maintainability of GUI test cases [66].

On average, ATUA generates 450 (one hour) and 2251 (five hours) fewer inputs than DM2 for each App version across all subject Apps. Since related work [3] has shown that roughly 35% of the updates concern the introduction of new features, under the assumption that inputs are uniformly distributed across updated features, we can estimate that ATUA generates, on average for each App version and across all subjects, 158 (one hour) and 788 (five hours) fewer inputs than DM2 for testing new features. Consequently, ATUA generates 292 (one hour) and 1463 (five hours) fewer inputs than DM2 for testing bug fixes and improved features (i.e., changes concerning non-functional requirements).

When testing new features, the output generated by each input should be manually verified; for example, by inspecting the screenshots of the GUI trees visualized after triggering an input (they are automatically captured by ATUA) and determining if they match the expected results. Unfortunately, the software engineering literature lacks studies about the cost of manual verification of GUI trees; assuming, for the sake of illustration, that visual inspection of GUI trees takes a few minutes, say ranging from one minute to five minutes, ATUA may lead to savings within the following intervals of [158-790] and [788-3940] minutes, respectively the for one-hour and five-hour test budgets. In the App development context, where Apps are frequently released (e.g., weekly or bi-weekly) and, additionally, test cases might need to be executed every day following continuous integration practices, such effort savings appear to be particularly beneficial, especially considering that testing should be performed by highly-trained engineers with a deep understanding of the App's features.

When testing updated features, engineers can re-execute the generated test input sequences on previous App versions in order to compare results and, ideally, eliminate oracle costs. However, we have to expect that a number of maintenance operations are required in order to adapt test sequences to a different App version. Pan et al. [66], for example, report that 26.5% of the test inputs need to be repaired. ATUA will thus save engineers from manually repairing 77 and 388 inputs, respectively for the one-hour an five-hour test budget. Under time pressure, which is the case when Apps are frequently released, this is a significant advantage.

Effectiveness ATUA is the approach that, on average, most effectively tests updated Apps within practical time budgets and human effort. For the one-hour budget, better from competing approaches, it exercises more than 60% of target methods and 50% of target instructions. With a five-hour budget, it exercises more than 70% of target methods and 60% of target instructions. Higher percentages can probably be reached with longer execution budgets, which were not possible in our context given the computational costs of our experiments. Based on these results, we can claim that ATUA can contribute to reducing development costs; indeed, engineers would then be able to focus their manual testing effort on a reduced portion of the developed App.

When comparing with other approaches, we observed that for both one-hour and five-hour budgets, on average, ATUA achieves method and instruction coverage results increased by at least 10% with respect to the second-best approach (DM2), a practically significant improvement. The effectiveness of ATUA is comparable to the effectiveness of DM2 and APE only when ATUA cannot fully leverage static analysis to determine the relation between inputs and WindowTransitions, i.e., when Apps integrate input handlers that are selected at runtime based on the nature of input data, which happens, for example, in the presence of WebViews. For the six subjects for which static analysis can effectively be exploited, the percentage of improvement rises above 8%. Among our subject Apps, in the worst case (i.e., five-hour budget), ATUA is comparable to other approaches for one third of the subjects and otherwise fares better; considering that (1) no single competing approach achieves similar coverage as ATUA for these three subject Apps (e.g., DM2 achieves the same results as ATUA for at most two), (2) competing approaches never outperform ATUA but at best reach the same effectiveness, ATUA remains the best choice.

Finally, ATUA has shown to be complementary to other approaches. Indeed, for both one-hour and five-hour budgets, it can exercise 518 and 478 target methods not covered by other approaches, three and six times the number of the second best approach. Thus, when combining testing approaches to cover higher target method coverage, ATUA should be included. Finally, as an explanation of the above results, we have observed that the three testing phases integrated in ATUA enable the generation of complex input sequences, specific App states, or diverse App settings that are required to test updated features.

3.4.7 Threats to Validity

We discuss internal, conclusion, construct and external validity according to standard practice [104, 105, 106].

Internal validity To address threats to *internal validity*, we should ensure that the observed outcome (inputs and code coverage, in our case) depends on the treatment (i.e., the test automation approaches) and not external factors (e.g., implementation errors and diverse experimental conditions) [104].

To minimize *implementation errors*, we have carefully inspected and tested ATUA before running our experiments. Also, for the state-of-the-art approaches, we relied on the software released by their authors, which had been used in several experiments.

To *ensure the same conditions* for all the experiments, we executed each tool on a clean instance of the same Android emulator with newly installed Apps. However, the same experimental conditions may not be guaranteed for Apps that depend on external data sources (e.g., to visualize news) [107]; indeed, in the presence of external data source, test results may depend on the content being visualized at a specific instant (e.g., the presence of a video in the latest news). Our case study subjects include six Apps loading external data (i.e., Nuzzel, Wikipedia, Yahoo weather, Wikihow, BBC Mobile, and Citymapper) because they are highly popular and representative.

To address this threat, taking advantage of our Grid infrastructure, for each test budget (i.e., one hour and five hours), for each subject App version, we executed all the testing tools in parallel in five batches with two sequential executions each. In practice, for each subject App's version, for each tool, we ran ten executions distributed over a time frame of two hours (for a one-hour budget) and ten hours (for a five-hour budget). Our experimental configuration should minimize the threat for Apps (i.e., Wikipedia, Wikihow, and Citymapper) loading remote content that unlikely changes in the time frame of our executions (i.e., ten hours max).

In addition, we believe that our configuration also addresses the threat for the remaining three Apps (i.e., Nuzzel, Yahoo weather, BBC Mobile) because, by running all the different testing tools in parallel, we maximize the likelihood of processing the same remote content (i.e., news or weather forecasts) when triggering the same Actions.

Conclusion validity Threats to *conclusion validity* concern the statistical power of our results, invalid statistical test assumptions, reliability of measurements, and random irrelevancies [105].

Since the underlying distribution of the data (i.e., code coverage achieved with test automation approaches) is not known in our context, for statistical significance, we rely on the Mann Whitney U-test, which has high *statistical power* for different underlying distributions, even for a small number of samples [108]. Also, to let the readers draw conclusions in context about the proposed approach, we report both p-values and effect sizes.

To avoid violating the *assumptions of parametrical statistical tests*, we rely on a non-parametric test and effect size measure (i.e., Mann Whitney U-test and the Vargha and Delaney's A_{12} statistics, respectively).

To ensure *reliability*, our measurements (i.e., code coverage) have been collected through widely used, open-source tools.

In our context, the only source of *random irrelevancies* might be the workload of the machines used to run the experiments, which may slow down the performance of some of the tools. To mitigate this threat, in addition to rely on a Grid infrastructure with guarantees for the provided service level, we manually inspected execution logs to exclude the presence of anomalies biasing results (e.g., exceptions due to the host environment).

Construct validity According to standard practice, we discuss *construct validity* in terms of face, content, convergent, and predictive validity [106]. The constructs considered in our work are effectiveness and cost. Effectiveness is measured through two reflective indicators, which are target method coverage and target instruction coverage. Cost is measured in terms of the number of inputs being generated, for reasons that were carefully discussed.

Face validity concerns the selection of appropriate reflective indicators. For effectiveness, we rely on method and instruction coverage, which is common practice [13, 107]. For cost, we measure the number of inputs being generated. At the beginning of Section 4.3, we have discussed that, in our context, the number of inputs is a good surrogate to enable the comparison of testing cost.

Content validity concerns the adequacy of reflective indicators to cover the breadth of the construct. We rely on code coverage since it has been recently shown that there is moderate to high correlation between code coverage and detection of real faults [102]. Also, code coverage is a necessary condition to uncover faults and, therefore, it remains a priority for test engineers. Concerning the breadth of the cost construct, in the introduction to this Section we have discussed that a direct and precise cost estimate can only be obtained with experiments involving engineers using the selected testing techniques in the field under controlled conditions, which we leave to future work.

Concerning *convergence*, we have computed the non-parametric Kendall's correlation coefficient, for all the pairs of reflective indicators, for each subject. Unsurprisingly, target method and target instruction coverage are highly correlated (i.e., $\tau \geq 0.7$, for all the subjects), which is expected for reflective indicators used to infer the same construct. Instead, a low correlation (i.e., $\tau < 0.35$, for all the subjects) is observed between inputs being triggered and target coverage, which is expected since these two reflective indicators are used for distinct constructs.

To address *predictive validity*, we reported statistics for all our research questions.

External validity To address threats to *external validity* we have considered nine popular Apps, downloaded thousands of times worldwide, that have been considered in the empirical evaluation of related work. Also, for each App, we considered up to ten App versions, based on their availability, for a total of 72 App versions tested. The considered Apps greatly vary regarding the overall number of lines of code and updated lines between versions. Because of their diversity, we believe our subjects to be representative of the Apps landscape.

To account for randomness, we tested each App version ten times with every testing tool considered; more than the usual practice of three to five repetitions [37]. Despite the high computational cost (17280 test execution hours, in total), this enabled us to derive solid statistical results for the comparison of different tools.

3.5 Conclusion

State-of-the-art App testing techniques are affected by two limitations: limited effectiveness (i.e., low code coverage) and the absence of automated oracles. To address the first limitation, given the high release frequency of Apps, we propose a solution (objective O1) to effectively focus the test budget on updated (i.e., modified and new) methods. In other words, within practical test execution time, we aim to maximize the coverage of updated methods and their instructions. To address the second limitation, we aim (objective O2) to generate a significantly reduced set of test inputs, compared to state-of-art approaches, thus proportionally saving the corresponding human effort required to visualize test outputs or correct test scripts.

To achieve the two objectives above, we developed ATUA, an automated App testing technique that integrates multiple analysis strategies. To achieve O1, it combines static analysis, to determine the inputs that execute updated features, and random exploration, to overcome the limitations of static analysis. To achieve O2, it relies on dynamically-refined state abstraction functions, to determine when distinct inputs lead to a same program state, and relies on information retrieval techniques, to identify dependencies among App features.

We performed an empirical evaluation where we compared ATUA with state-of-the-art approaches implementing testing strategies based on dynamically derived models (DM2), random exploration (Monkey), and dynamic state abstraction (APE). For our experiments, we considered practical execution time budgets of one and five hours, corresponding respectively to approximate time constraints in the context of continuous integration and overnight testing. Concerning human effort (objective O2), ATUA is the approach that generates the smallest set of inputs with the highest coverage per input. ATUA, on average across subject Apps, saves around 32.6% of the effort, compared to the second-best approach (DM2). Further, it exercises 38.5% more instructions than DM2 per input. Differences with APE and Monkey are much larger. Concerning effectiveness within time budget (objective O1), on average, ATUA automatically exercises up to 70% of updated methods and 60% of instructions belonging to updated methods, 6% more than the second best approach (i.e., DM2). These results show that the analysis strategies integrated in ATUA can drive testing towards an efficient use of the test budget (execution time and effort), thus providing clear benefits when upgrading and testing an App.

Chapter 4

CALM: Continuous Adaptation of Learned Models

4.1 Introduction

Software applications for mobile devices (i.e., Apps) are updated frequently, mainly to improve the user experience and fulfill marketing strategies [3, 1, 2]. Our industry partners highlighted that in such scenario, where the time dedicated to development and testing is limited, it is important to focus testing effort on the features that have been modified and introduced in the new App version.

Unfortunately, automated App testing techniques do not target updated features but exercise whole Apps and cover their implementation only partially (e.g., they exercise around half of the App methods [14, 13]). When coverage is limited, regression test selection techniques [16, 15] are unlikely to help engineers in selecting test cases that exercise the updated features. Therefore, the automated testing of updated features remains an open problem. Further, existing techniques detect only crashes or data loss [24] though a recent study on functional faults affecting Android Apps reports that 95% of the failures likely require visual inspection to be detected. Among these, content related issues account for 21%, structure related issues 40%, incorrect interaction 19%, and functionality not taking effect 12%) [31]. Unfortunately, visual inspection of App outputs is practically infeasible when automated testing tools generate a large number of test inputs, each one leading to a new output screen to be inspected.

In the previous chapter, we have shown that static and dynamic program analyses drive model-based App testing towards maximizing the coverage of updated methods while using a limited number of test inputs [33]. We named our previous approach ATUA; for a same number of App screens to be exercised, it outperforms state-of-the-art (SOTA) approaches in terms of code coverage.

Although ATUA demonstrated to be more effective than approaches not focused on App updates, it does not reuse App models across versions, which makes the test process inefficient (e.g., for every App version, it may resort to random exploration to trigger Window transitions not identified by static analysis). In the literature, inferred models have been reused to repair test scripts [109], execute test cases on different platforms [19, 18], and automate regression testing [110]. Unfortunately, the only approach reusing models across versions is Fastbot2 [46], a recent approach that reuses a probabilistic model learned in a previous version to drive testing in a newer version. However, our empirical results (see Section 4.3) show that, since it does not integrate static analysis, it cannot effectively target updated features.

We present *Continuous Adaptation of Learned Models (CALM)*, an App testing technique that efficiently tests updated Apps by relying on models learned with previous App versions. CALM leverages ATUA to select test inputs that exercise updated methods. However, CALM improves over ATUA by combining dynamic and static program analysis to adapt and improve the model learned when testing a previous App version. The reuse of an existing model enables CALM to efficiently use the test budget to exercise updated methods rather than to determine, with random exploration, how to reach Windows already reached in previous App versions.

Before testing a new App version, CALM relies on static program analysis to identify changes in the App GUI that should be reflected in the App model. This includes, for example, removing state transitions triggered by Widgets no longer present in the updated App. In addition, it integrates heuristics for the runtime adaptation of App models to make model reuse effective. Precisely, it introduces *layout-guarded abstract transitions* to deal with non-determinism; it derives *probabilistic Action sequences* to deal with *state explosion*; it detects model states that are new but compatible with previously executed Action sequences (i.e., *backward-equivalent*); it relies on *online and offline model refinement* to identify and remove obsolescent model states. Finally, CALM identifies and provides engineers with only the output screens rendered by the App after an updated method had been exercised, thus greatly minimizing test oracle costs.

Our empirical evaluation shows that, for a one-hour test budget, CALM exercises a significantly larger percentage of updated methods and instructions than SOTA tools (ATUA, Monkey [42], APE [37], Fastbot2, TimeMachine [111], and Humanoid [56]). For a same maximum number of screen outputs to be visualized, CALM outperforms the second-best SOTA approach by 6 percentage points (pp). Most importantly, this difference keeps increasing with the test budget and is even larger (13 pp) for quick test sessions with updates of small size, which are by far the most frequent.

4.2 Proposed Approach: CALM

CALM supports engineers in testing updated Apps by relying on App models that are incrementally constructed and adapted, version after version. Similar to ATUA, CALM aims to exercise all the *target methods* (see Section 3) of an App version. For the first version of the AUT, CALM treats each method as a target method, and starts from an empty App model. For every App version following the first, CALM relies on the App model produced for the previous App version (*base App*).

CALM works in four steps, shown in Figure 4.1. In Step 1, CALM relies on an extension of RCVDiff to compare the EWTGs generated by ATUA's Extended GATOR for the base and updated App.

In Step 2, CALM relies on the identified differences to generate an updated App model by adapting the EWTG and the DSTG of the base App model. By reusing the DSTG generated

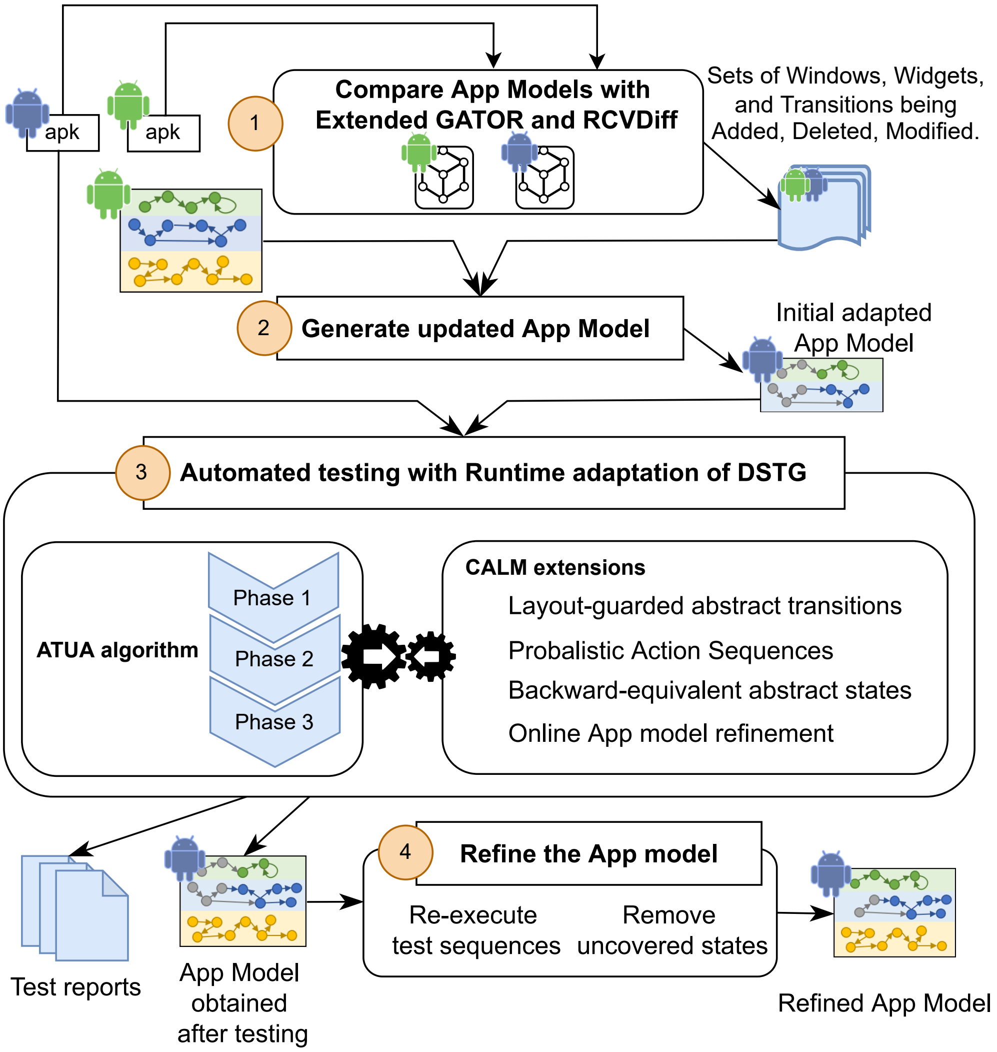
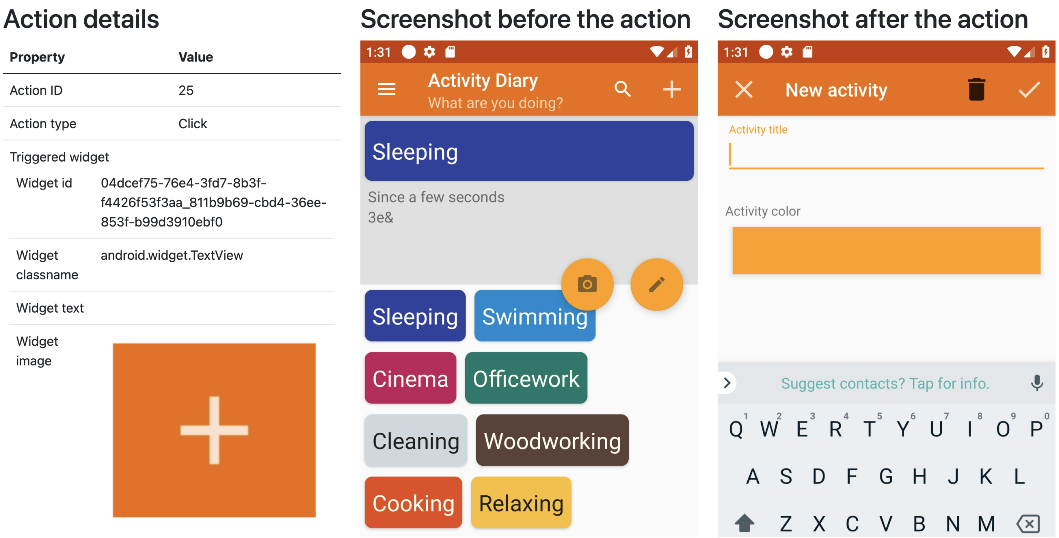

apk
apk
1
Compare App Models with Extended GATOR and RCVDiff
Sets of Windows, Widgets, and Transitions being Added, Deleted, Modified.
2
Generate updated App Model
Initial adapted App Model
3
Automated testing with Runtime adaptation of DSTG
ATUA algorithm
Phase 1
Phase 2
Phase 3
CALM extensions
Layout-guarded abstract transitions
Probalistic Action Sequences
Backward-equivalent abstract states
Online App model refinement
4
Refine the App model
Re-execute test sequences
Remove uncovered states
Test reports
App Model obtained after testing
Refined App Model

Figure 4.1: CALM App testing process

for the base App, CALM aims to optimize testing efficiency by minimizing the effort spent to generate AbstractStates and AbstractTransitions for the updated App. CALM does not inherit the GSTG because the GUITrees in the GSTG are used to refine AbstractStates (see Section 3.2) but inherited GUITrees might be outdated thus leading to invalid AbstractStates.

In Step 3, CALM relies on an extended version of the ATUA testing process to test the updated App. CALM's extensions maximize the effectiveness of model reuse by enabling the execution of Action sequences derived from previous App models, even when the AbstractStates observed in the two versions present differences. Further, CALM updates the App model to reflect the actual behaviour of the AUT.

The output of Step 3 is an App model for the updated App version. CALM generates a report with a set of triples <GUI screenshot, target action, GUI screenshot> reporting



Action details

Property	Value
Action ID	25
Action type	Click
Triggered widget	
Widget id	04dcef75-76e4-3fd7-8b3f-f4426f53f3aa_811b9b69-cbd4-36ee-853f-b99d3910ebf0
Widget classname	android.widget.TextView
Widget text	
Widget image	

Figure 4.2: Example of action output provided to the end-user

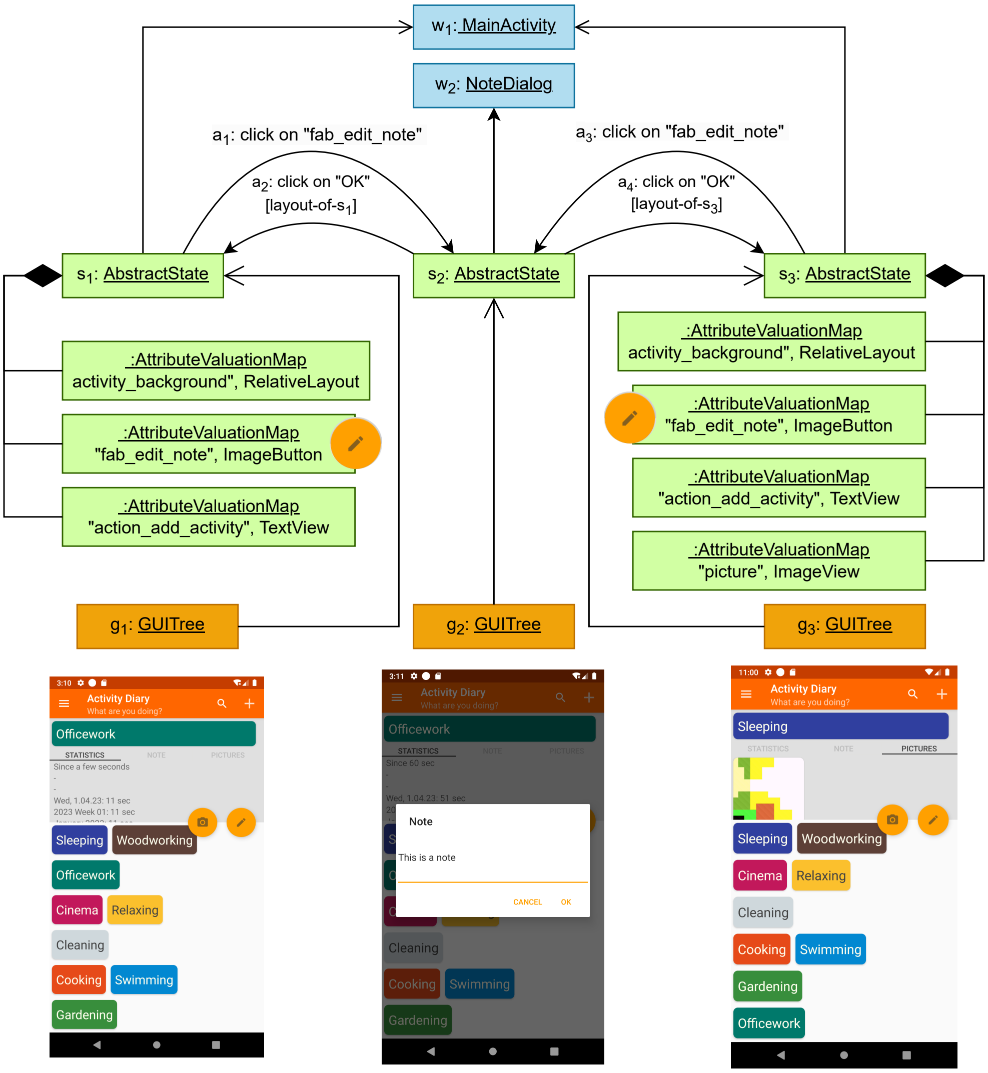
for every target Action (i.e., action triggering the execution of target methods) triggered by CALM the screenshot before and after the execution of the action. An example is shown in Figure 4.2. To avoid wasting engineers' time, only actions that increase code coverage are reported; we refer to such actions as *Unique Target Actions* (*UTAs*). The generated triples support crowdsourcing-based oracles (e.g., they can be shared among a set of App users to determine if the output is functionally correct or not [80]). Further, engineers can also visualize, from the GSTG, the sequence of inputs and outputs terminating with the triple shown in the repo

In Step 4, after testing, CALM refines the App model to eliminate infeasible paths due to AbstractStates that are unreachable. Those are either AbstractStates from the base App model not observed when testing the updated App or AbstractStates introduced when testing the updated App but becoming quickly obsolete. The output of Step 4 is a refined App model to be used when testing the next App version.

In the following Sections, before detailing the CALM steps, we first discuss how model reuse can exacerbate state abstraction's limitations and reduce testing effectiveness.

4.2.1 Limitations of state abstraction

During testing, CALM, like ATUA, derives a sequence of Actions to be triggered to reach a target Window (in Phase 1) or a target AbstractState (in Phases 2 and 3). Such sequence


w1: MainActivity
w2: NoteDialog
a1: click on "fab_edit_note"
a3: click on "fab_edit_note"
a2: click on "OK" [layout-of-s1]
a4: click on "OK" [layout-of-s3]
s1: AbstractState
s2: AbstractState
s3: AbstractState
:AttributeValuationMap activity_background", RelativeLayout
:AttributeValuationMap "fab_edit_note", ImageButton
:AttributeValuationMap "action_add_activity", TextView
:AttributeValuationMap activity_background", RelativeLayout
:AttributeValuationMap "fab_edit_note", ImageButton
:AttributeValuationMap "action_add_activity", TextView
:AttributeValuationMap "picture", ImageView
g1: GUITree
g2: GUITree
g3: GUITree
Activity Diary
What are you doing?
Officework
STATISTICS
NOTE
PICTURES
Since a few seconds
Wed, 1.04.23: 11 sec
2023 Week 01: 11 sec
Sleeping
Woodworking
Officework
Cinema
Relaxing
Cleaning
Cooking
Swimming
Gardening
Note
This is a note
CANCEL
OK
Since 60 sec
Wed, 1.04.23: 51 sec
Sleeping
Woodworking
Cinema
Relaxing
Cleaning
Cooking
Swimming
Gardening
Officework

Note: The dialog window's background is not captured by our $\mathcal{L}_*$; therefore, the dialogs shown after g_1 and g_3 belong to the same AbstractState (i.e., s_2)

Figure 4.3: Closing a dialog brings the App back to the same screen where the dialog was opened.

also specifies the AbstractState that is expected after every action; if the expected state is not reached after a certain Action (e.g., because of non-determinism) the rest of the sequence is not executed. Indeed, it makes no sense to execute Actions whose preconditions (e.g., a visible Widget) do not hold. When such state mismatch is observed, CALM derives a new Action sequence that reaches the target Window/AbstractState from the current AbstractState. Unfortunately, when models are reused, state abstraction mechanisms often lead to such state mismatches. Below, we describe four scenarios that we address in CALM Steps 3 and 4.

First, state abstraction may lead to *non-deterministic AbstractTransitions* when the effect of an Action depends on a previously performed Action. Usually, such non-determinism is observed when AbstractTransitions bring the App into an AbstractState recently visited or into an AbstractState derived from a recently visited one. The latter often consists of an AbstractState having the same layout but a different number of Widgets than the previous AbstractState. In both those scenarios, the previous state may be different at every execution of the AbstractTransition. An example based on Activity Diary is shown in Figure 4.3: the action of clicking on the *Close dialog* button in state s_2 brings the App to the same screen (and same AbstractState) visualized before opening the dialog (i.e., either s_1 or s_3). Non-deterministic abstract transitions are more likely with reused models because testing is started with a populated DSTG although the App is freshly started. For example, in the case of Figure 4.3, the App starts in s_1 and test automation may try to perform the infeasible sequence $\langle a_1, a_4 \rangle$ because shorter than the feasible sequence reaching s_3 (not shown in Figure 4.3).

Second, state abstraction may lead to *state explosion due to hidden Widgets*. It happens, for example, in the presence of Windows with Widgets that are dynamically added but keeping the behaviour of unmodified Widgets unchanged. Such state explosion limits the effectiveness of model reuse because, although the DSTG includes AbstractTransitions capturing the effect of Actions on a Widget (e.g., a button click on a drawer item triggering a WindowTransition), triggering such AbstractTransitions requires reaching the exact AbstractState in which the Widget was tested previously, which entails triggering several, potentially unnecessary, Actions.

Third, in the presence of modified Windows, AbstractStates may not match across versions although they are the source of AbstractTransitions that behave the same across versions. We call such AbstractStates *backward-equivalent AbstractStates*; they are observed in the presence of minimal changes in App Windows (e.g., few Widgets being added or removed). For example, if the updated App introduces a reset button for a Window with a form, an Action sequence exercising the submit button, which has not been modified, should remain executable in the updated version; however, the AbstractState of the two Window versions does not match because of the different number of Widgets in them.

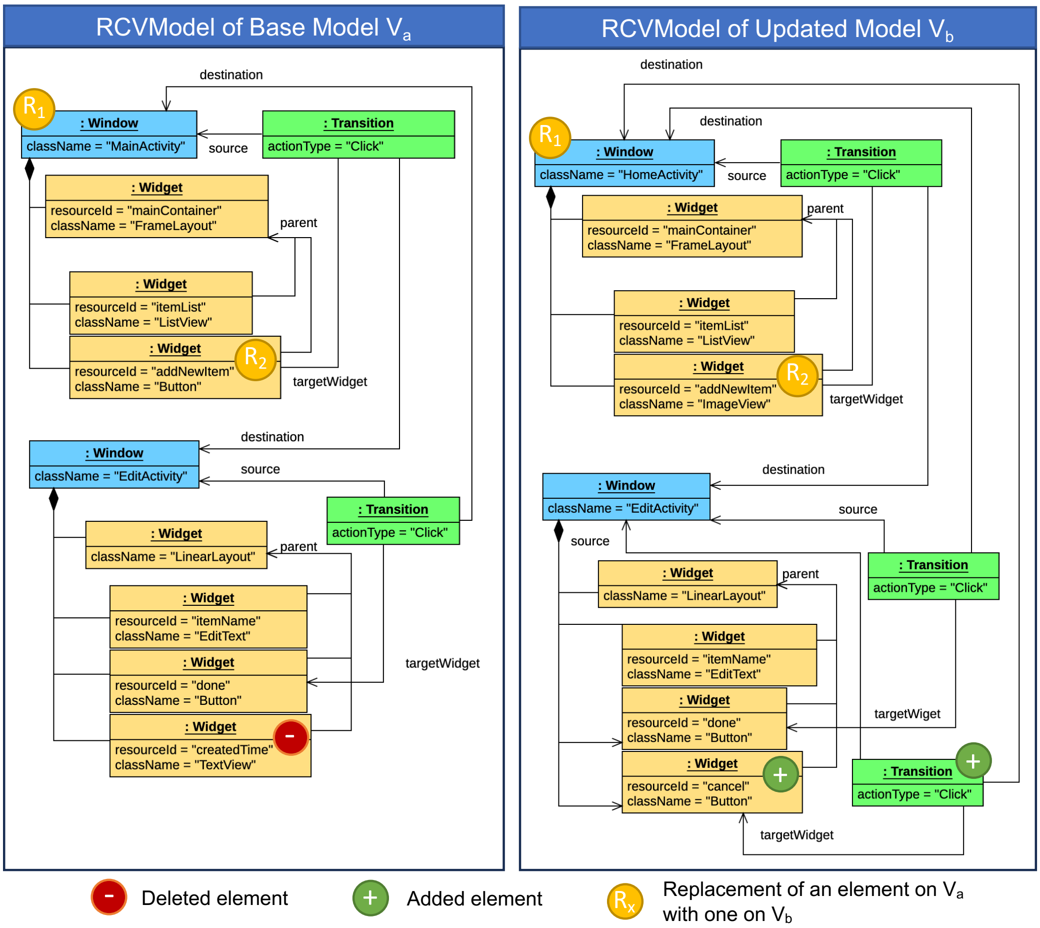
Fourth, we may observe *obsolete AbstractStates*. Indeed, AbstractStates often depend on a remote component (e.g., a news server) that provides data that change over time. Consequently, such AbstractStates become obsolete quickly (e.g., after a few minutes). Other

AbstractStates become obsolete because the behaviour of the updated App changed (i.e., it's not possible to reach a certain AbstractState with the same input sequence observed in a previous App version).

4.2.2 Step 1: Detect EWTG Differences

CALM compares two EWTGs by relying on RCVDiff. To this end, it generates an RCV-Model instance that captures the EWTG elements. To identify differences, we extended the RCVDiff algorithm as follows. First, our RCVDiff extension looks for elements (Window, Widget, Transition) that present matching attributes and references. Second, to determine what elements of the base App model had been replaced in the updated App model, our RCVDiff extension looks for additional elements that may *correspond* (e.g., because of a class renaming) by relying on the following:

- Since a Widget could be moved to another container (i.e., its parent changed), two Widgets correspond when all their attributes, except *parent*, match. Also, since a Widget could be replaced with another one implementing similar features (e.g., a button replaced by a clickable image), they correspond when all their attributes, except *className*, match.
- To correspond, two Windows should extend the same Window Type and other properties should match (e.g., class).
- To correspond, two Transitions should start from matching sources (i.e., Windows) and trigger the same action on a matching Widget (e.g., a Button). When the destination does not match, the updated transition simply reflects a change in the App behaviour.
- To correspond, element attributes should have high string similarity; precisely, we rely on the Levenshtein ratio with a threshold of 40%. The chosen threshold has been empirically demonstrated to be appropriate [112], as it enables handling cosmetic changes (e.g., fixing typos in labels). For XPaths, which capture the position of a Widget in the containing Window, we use a token-based distance: we split each string into tokens (separator is '/') and compute the cosine similarity distance [113] between the two token sets.



Note: Our RCVDDiff extension reports that the Windows named *MainActivity* in V_a and *HomeActivity* in V_b correspond because they have the same type and their other properties match (they are not listed in the Figure); it indicates that the Window has been renamed. CALM thus correctly records that the Window *MainActivity* in V_a has been replaced by Window *HomeActivity* in V_b. Further, our RCVDDiff extension reports that the Widget named *addNewItem* in V_a corresponds to the homonymous widget in V_b because all their attributes except their class name match; since the *className* attribute of the widget *addNewItem* has been changed from *Button* to *ImageView*, it indicates that a button Widget has been replaced by an image. CALM thus records that the Widget *addNewItem* in V_a has been replaced by the Widget *addNewItem* in V_b. Finally, RCVDDiff detects that, in the *EditActivity* Window, a Widget has been deleted (i.e., the TextView named *createdTime*), while a Widget (i.e., the Button named *cancel*) and a transition triggered by it have been added.

Figure 4.4: An example of RCVDiff Model of EWTGs belonging to two App versions.

CALM processes the RCVDiff output to identify Windows, Widgets, and Transitions being added, removed, or replaced; replaced elements are elements of the base App model with a corresponding element in the updated App model. Figure 4.4 shows an example output.

4.2.3 Step 2: Generate an Updated App Model

CALM performs four tasks to create the updated App model to be used when testing the updated App: (1) copies the base App model, (2) removes all the GSTG elements, (3) replaces the base EWTG with the updated EWTG, and (4) updates the DSTG. Since the first three activities are straightforward, below, we describe how CALM updates the DSTG.

CALM removes from the DSTG all the items associated to the elements deleted from the EWTG, which are: all the AbstractStates associated with deleted Windows, all the AVMs associated with deleted Widgets, all the AbstractTransitions associated with deleted Transitions. Further, it removes all the elements that become disconnected from the rest of the DSTG.

Window replacements are caused by Windows renaming; therefore, CALM assigns the replaced Window's AbstractStates to the replacing Window. Similarly, we assign replaced Widgets' AVMs to replacing Widgets. For Transition replacements, since they indicate a change in the source or destination Window but there is no mean to determine the mapping to an AbstractState for that Window, we simply remove the AbstractTransitions associated to the replaced Transition.

Added elements do not lead to any update of the DSTG because they were not present in the base App.

Figure 4.5 demonstrates how CALM integrates the DSTG of a base App model into an updated App model by relying on the information provided by the RCVDiff model (i.e., the one in Figure 4.4, in this example). In the updated App model, the AttributeValuationMap *createdTime* (i.e., *avm5*) is removed from the AbstractState s_2 because, in the base App model, avm5 was associated with the Widget w_7, which has been removed from the updated App. In the updated App model, the AbstractState s_1 is reassigned to the *HomeActivity*

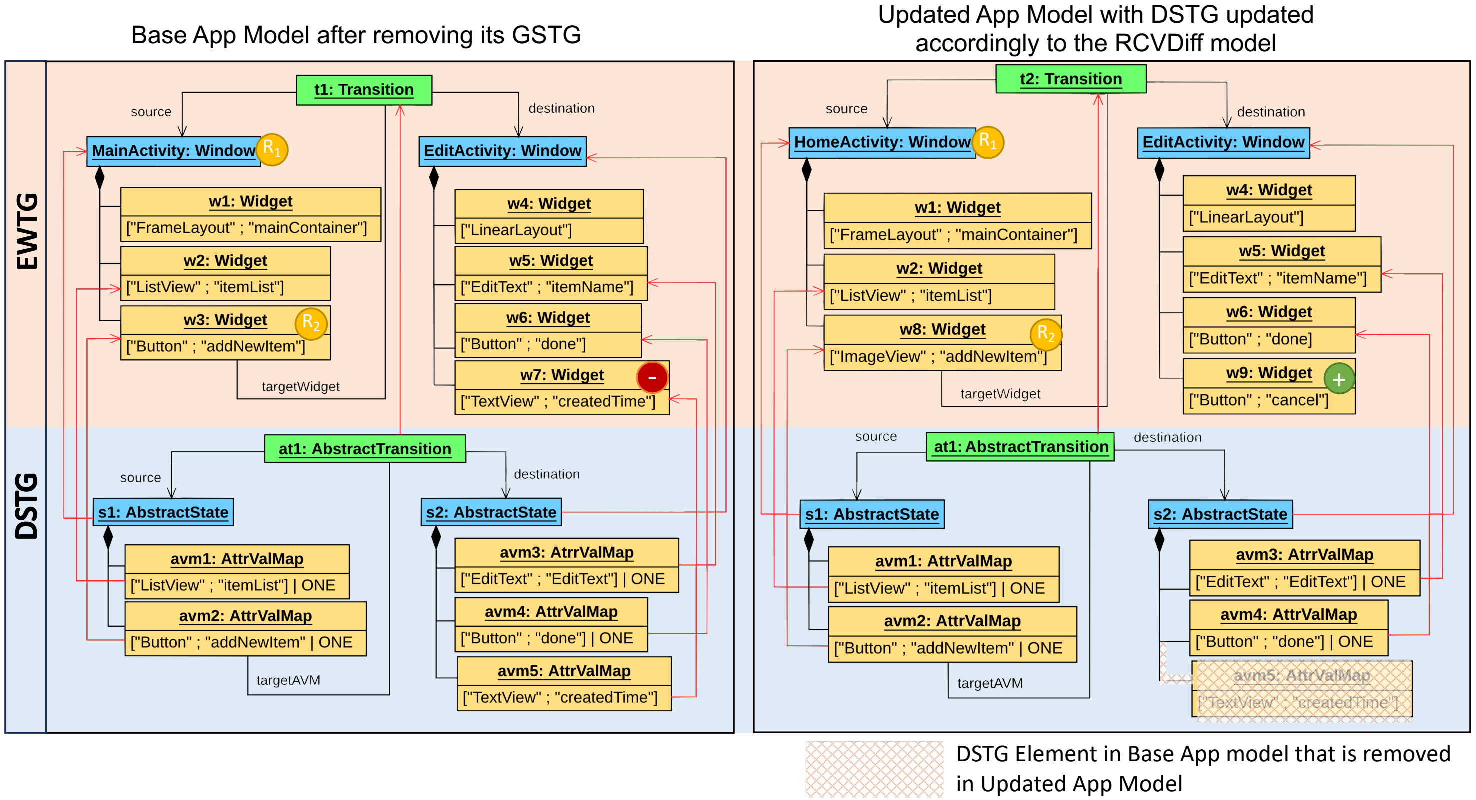


Figure 4.5: Illustration of how DSTG of Base App model is adapted in Updated App model accordingly to the RCVDiff model in Figure 4.4

Window because the *HomeActivity* Window replaces the *MainActivity* Window of the base App model. Still in the updated App model, the AttributeValuationMap *addNewItem* (i.e., *avm2*) is reassigned to the widget w_8, which has type *ImageView*; such change depends on w_8 being a replacement for Widget w_3, which is of type *Button*. Please note that preserving AVMs enable CALM to preserve the AbstractTransitions at_1, which brings the App to the AbstractState $s2$ from $s1$ when clicking on the *addNewItem* widget. Finally, since Widget w_9 was added to the updated App model, it has no DSTG element assigned to it. The other DSTG elements in the updated App model remain associated to the same elements in the base App model; or, more precisely, they are associated to EWTG elements of the updated App model that match the ones in the base App Model.

4.2.4 Step 3: Automated Testing with Runtime DSTG Adaptation

In this Section, we describe the solutions integrated into the testing Step to overcome the limitations described in Section 4.2.1. We aim to avoid deriving Action sequences that expect traversing AbstractStates that, in the updated App, are unreachable with such sequences.

Layout-guarded abstract transitions

To handle non-determinism due to AbstractTransitions bringing the App into a recently visited AbstractState or a state derived from a recently visited one, CALM augments App models with guard conditions specifying if the state reached by the transition is expected to be derived from a previously visited AbstractState. We call such transitions *layout-guarded abstract transitions*. During testing, before adding a state transition to the App model, CALM verifies if the destination state has a layout similar to the layout of any previously visited AbstractState; for that, it processes the GSTG backward till it reaches the initial state or a GUITree with an AbstractState having a layout similar to the destination state. To determine if two AbstractStates have a similar layout, CALM focuses on the AttributeValuations derived using the reducers belonging to $\mathcal{L}_1$, which do not include text and Widget children; indeed, text and Widget children are likely to vary when AbstractStates are derived from previously visited ones. Two AbstractStates have a similar layout if they share 80% of such AttributeValuations; we propose a threshold of 80% because it led to the best results (highest coverage of modified methods and instructions belonging to modified methods) in a preliminary experiment conducted with the latest version of all the subject Apps included in our empirical evaluation.

When generating Action sequences, CALM traverses *layout-guarded* AbstractTransition only if the referenced layout is similar to the layout of an AbstractState previously visited. In Figure 4.3, the transition from s_2 to s_3, triggered by the Action a_2, is guarded by $layout\_of(s_1)$. Thanks to such guard, at runtime, CALM will not suggest the infeasible sequence $\langle a_1, a_3 \rangle$ when being in s_1 after starting the App. Indeed, their AVMs derived with $\mathcal{L}_1$ differ because s_3 includes a picture Widget that is not present in s_1.

Probabilistic Action sequences

Often, in the presence of *state explosion due to hidden Widgets*, a sufficient condition to execute a whole Action sequence is the presence of all the Widgets targeted by the Actions in the sequence. In Figure 4.6, the sequence $\langle$*click on* $w1$, *click on* $w2$, *click on* $w3\rangle$ derived from the AbstractState S_6 might be feasible even if S_9 is observed. Therefore, an Action sequence should be selected by identifying the Actions leading to App screens that likely contain the Widgets targeted by the next Action in the sequence, till a Window with the targeted Input is reached.

Because of the observation above, CALM derives *probabilistic Action sequences* in addition to *deterministic Action sequences* (i.e., what is derived by ATUA). For every Action, they capture the likelihood of reaching an App screen that includes the Widget required by the next Action. When traversing the DSTG to derive an Action sequence, CALM adds a

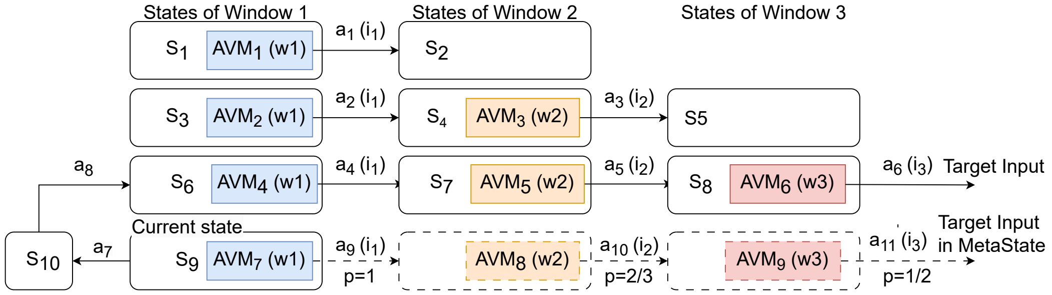


Figure 4.6: Part of a DSTG with a Probabilistic Action sequence; MetaStates and MetaTransitions are dashed.

transition for every Input i that has never been exercised in the current AbstractState. We call such additional transitions *MetaTransitions*. The destination of a MetaTransition is a *MetaState*; a MetaState tracks all the Widgets ever visualized in any App screen after exercising the Input i. To derive MetaStates, CALM considers the AbstractStates reached after exercising i and identifies all the Widgets belonging to them. Each MetaTransition leaving a MetaState is associated with a probability of being available (i.e., it captures the likelihood that the Widget targeted by the MetaTransition is interactable in any of the observed AbstractStates). In Figure 4.6, Input i_1, which targets Widget w_1 in Window wi_1, may bring the App into three distinct AbstractStates, two of which including widget w_2; the probability of exercising w_1 is 1.0 (w_1 is present in the current state S_9), the probability of reaching w_2 after exercising i_1 is 0.67 (i.e., $2/3$). Similarly, the probability of exercising the target input on $w3$ is 0.5.

During testing, CALM should select the Action sequence that brings the App into the target Window/AbstractState with a minimal cost. However, MetaTransitions may not bring the App into a desired MetaState, and thus it may not be feasible to fully exercise a probabilistic Action sequence. To maximize efficiency, we should estimate an Action sequence cost by accounting for the risk of not reaching a desired MetaState.

We assume that *deterministic Action sequences* are likely to be fully executed because they do not include MetaTransitions; therefore, their cost depends on the time required to execute all their Actions. Given a *probabilistic Action sequences* and a *deterministic Action sequence* with the same length, CALM should exercise the deterministic one because it does not present feasibility risks. Consequently, to account for feasibility risks, when computing the cost of a probabilistic input sequence ϕ, we heuristically sum the cost of executing the whole sequence ϕ with the cost of executing a partial subsequence of ϕ multiplied by the likelihood of not executing the whole sequence ϕ:

$$cost(\phi) = cost\_full(\phi) + cost\_partial(\phi) * likelihood\_partial(\phi)$$

The cost of executing a whole Action sequence ϕ (either partial or deterministic) is the sum of the cost of executing all its n Actions:

$$cost\_full(\phi) = \sum_{j=1}^{n} cost(action_j)$$

Since we empirically observed that all the Actions, except App reset, take almost the same time to execute, and App reset takes around ten times the other Actions, we assign to reset Actions a cost of 10, and 1 to other Actions.

The cost of executing only part of an Action sequence depends on the number of Actions being triggered; since we cannot predict how many Actions will be executed, we conservatively assume that, on average, half of the Actions of *probabilistic input sequences* are executed, which leads to:

$$cost\_partial(\phi) = \frac{\sum_{j=1}^{n} cost(action_j)}{2}$$

Finally, the likelihood of executing an input sequence depends on the likelihood of observing all the Widgets targeted by the Actions in the sequence. Therefore, the probability of partially executing an input sequence ϕ is:

$$likelihood\_partial(\phi) = 1 - \prod_{j=1}^{n} p(action_j | s_{j-1})$$

with $p(action_j | s_{j-1})$ being the probability of observing the Widget required to trigger the Action $action_j$ in the state reached by $action_{j-1}$. If s_{j-1} is not a MetaState, we expect the widget to be available; therefore, $p(action_j | s_{j-1}) = 1.0$. If s is a MetaState, $p(action_j | s_{j-1})$ matches the *MetaTransition probability* described above. For a deterministic input sequence ϕ_d, since input widgets are available in the reached AbstractStates (i.e., $p(i_j | s_{j-1}) = 1$), we have:

$$likelihood\_partial(\phi_d) = 0$$

During testing, when executing a partial input sequence ϕ, for each MetaState s_e expected after an Action i_j, CALM verifies that the Widget to be triggered next is available; otherwise a new input sequence needs to be selected.

Based on the above, in the example in Figure 4.6, the probabilistic Action sequence $\langle a_9, a_{10}, a_{11}\rangle$ will be selected instead of the deterministic Action sequence $\langle a_7, a_8, a_4, a_5, a_6\rangle$, thus reducing the number of Actions required to reach the test target. Indeed, since the current state is S_9 and the objective is to reach $w3$ and trigger i_3, the cost of the probabilistic action sequence $\langle a_9, a_{10}, a_{11}\rangle$ would be computed according to the following equations:

$$\begin{cases} cost(\langle a_9, a_{10}, a_{11}\rangle) = cost\_full(\langle a_9, a_{10}, a_{11}\rangle) \\ \qquad + cost\_partial(\langle a_9, a_{10}, a_{11}\rangle) * likelihood\_partial(\langle a_9, a_{10}, a_{11}\rangle) \\ cost\_full(\langle a_9, a_{10}, a_{11}\rangle) = cost(a_9) + cost(a_{10}) + cost(a_{11}) = 1 + 1 + 1 = 3 \\ cost\_partial(\langle a_9, a_{10}, a_{11}\rangle) = \dfrac{cost(a_9) + cost(a_{10}) + cost(a_{11})}{2} = 1.5 \\ likelihood\_partial(\langle a_9, a_{10}, a_{11}\rangle) = 1 - \big(p(a_9) * p(a_{10}) * p(a_{11})\big) \\ \qquad = 1 - (1 * 2/3 * 1/2) = 0.66 \end{cases} \tag{4.1}$$

They lead to:

$$cost(\langle a_9, a_{10}, a_{11}\rangle) = 3 + (1.5 * 0.66) = 3.99 \tag{4.2}$$

The Action sequence $\langle a_7, a_8, a_4, a_5, a_6\rangle$, instead, leads to:

$$\begin{aligned} cost(\langle a_7, a_8, a_4, a_5, a_6\rangle) &= cost\_full(\langle a_7, a_8, a_4, a_5, a_6\rangle) \\ &+ cost\_partial(\langle a_7, a_8, a_4, a_5, a_6\rangle) * likelyhood\_partial(\langle a_7, a_8, a_4, a_5, a_6\rangle) \\ &= \big((cost(a_7) + cost(a_8) + cost(a_4) + cost(a_5) + cost(a_6)\big) \\ &+ cost\_partial(\langle a_7, a_8, a_4, a_5, a_6\rangle) * likelyhood\_partial(\langle a_7, a_8, a_4, a_5, a_6\rangle) \\ &= 5 + cost\_partial(\langle a_7, a_8, a_4, a_5, a_6\rangle) * 0 \\ &= 5 \end{aligned} \tag{4.3}$$

Backward-equivalent abstract states

A *backward-equivalent AbstractState* differs from an AbstractState expected by the input sequence and inherited from the base App's DSTG, but enables performing the same actions.

Precisely, an AbstractState s_o observed in the updated App is backward-equivalent to a state s_e derived from the base DSTG when:

- s_e and s_o are associated to the same Window; otherwise, they cannot be equivalent because different Windows implement different features.
- Every AVM in s_e matches an AVM in s_o. Otherwise, it would not be possible to trigger, in s_o, the same Actions triggerable in s_e.
- Every AVM in s_o, except the ones for the EWTG Widgets added or replaced in the updated App, matches an AVM in s_e. If this condition does not hold, a Widget may have different AVMs in the two App versions (e.g., a checkbox is no longer checked). In such case the updated App changed its behaviour and, consequently, a same Action may not exercise the same methods in the base and updated version.

For example, taking the AbstractState s_2 in Figure 4.5, which is inherited from the base App Mode, as an expected AbstractState, CALM observes a new AbstractState s_3 similar to s_2 but including additionally an Attrib uteValuationMap representing the added EWTG Widget w_9 in the "EditActivity" Window. Since there is only a mismatch between s_3 and s_2, which is related to the added EWTG Widget, s_3 is backward-equivalent to s_2. This implies that any required action that needs to be performed on s_2 could be done on s_3 too.

Online App model refinement

Online model refinement aims at determining if expected states that are not observed when exercising an Action sequence are obsolete. It is necessary because, otherwise, CALM may keep selecting Action sequences that include an unreachable state, which leads to a waste of resources.

When exercising an Action sequence, if the state expected after the i^{th} action (s_{e_i}) does not match (or is backward-equivalent to) the observed state (s_{o_i}), CALM determines if s_{e_i} is obsolete. If s_{e_i} has already been observed when testing the updated App, then it is not obsolete; therefore, s_{o_i} is the result of nondeterminism and CALM applies ATUA's procedure to minimize nondeterminism (i.e., it refines $s_{e_{i-1}}$ using $\mathcal{L}$). Otherwise (i.e., if

s_{e_i} had not been observed with the updated App), CALM removes from the DSTG the AbstractTransition connecting $s_{e_{i-1}}$ with s_{e_i}.

4.2.5 Step 4: Refine the App Model Offline

After testing an App version V_x, *to further clean up the App model from unreachable AbstractStates*, CALM removes from the App model all those AbstractStates that were not visualized although belonging to exercised Windows.

Further, *to remove AbstractStates that become quickly obsolete*, after testing an App version V_x, CALM re-executes, offline, the sequences of test inputs captured by the GSTG. During such re-execution, if a test input does not bring the App into the expected AbstractState s_e, then CALM annotates s_e as obsolete. When testing version V_{x+1}, to avoid wasting the test budget, CALM does not generate input sequences that traverse obsolete states.

Finally, since we empirically observed that Windows with obsolete states often present newer states that quickly become obsolete (e.g., Apps that display updated news every few minutes), CALM considers obsolete any AbstractState identified when testing V_{x+1} but not reachable with Action sequences traversing their incoming transitions, while still testing V_{x+1}. Our approach enables CALM to rely on obsolescent AbstractStates till they become obsolete.

4.3 Empirical Evaluation

We performed an empirical evaluation that aims to address the following research questions (RQs):

- *RQ1. Is CALM more effective than competing approaches in testing App updates, for a same test budget?* We aim to determine if CALM performs significantly better (code coverage) than ATUA and SOTA approaches that complement ATUA [33].
- *RQ2. How do CALM and competing approaches fare, for different testing time budgets, with updates of different magnitude?* Updated Apps may need to be tested quickly (e.g.,

Table 4.1: Selected subject systems.

App	V0	V1	V2	V3	V4	V5	V6	V7	V8	V9
AD	105	111	115	118	122	125	130	131	134	
BM	5.1.0	5.10.0	5.11.0	5.12.0	5.13.0	5.4.0	5.5.0	5.6.0	5.8.1	5.9.0
CM	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9	10.0
FM	44	53	77	79	82					
WI	198	10239	10263	10264	10269					
VP	3.1.4	3.1.5	3.1.7	3.2.12	3.2.2	3.2.3	3.2.6	3.2.7	3.2.9	
YM	1.16.0	1.16.1	1.16.2	1.17.3	1.18.1	1.19.1	1.20.1	1.20.3	1.20.5	1.20.7

Apps: AD: Activity Diary, BBC: BBC Mobile, CM: Citymapper, FM: Amaze File Manager, WI: Wikipedia, VP: VLC Player, YM: Yahooweather Mobile

Table 4.2: Number of updated methods for each App version. For V0 (assumed as the initial version), we report all the methods of the App.

App	V0	V1	V2	V3	V4	V5	V6	V7	V8	V9
AD	260	18	3	12	117	39	28	1	49	
BM	10706	649	27	44	25	603	242	553	77	95
CM	9629	51	37	55	73	119	76	73	12	69
FM	2042	306	415	11	644					
WI	7477	1430	535	13	94					
VP	6796	672	26	3	149	13	51	33	42	
YM	2932	5	4	243	10	16	118	101	12	9

Apps: AD: Activity Diary, BBC: BBC Mobile, CM: Citymapper, FM: Amaze File Manager, WI: Wikipedia, VP: VLC Player, YM: Yahooweather Mobile

after each code commit, with a limited test budget). However, both the magnitude of the update (e.g., number of updated methods) and the testing time budget may affect the performance of CALM. Therefore, we study how the effectiveness of CALM compares with competing approaches over time and for updates of different magnitude.

4.3.1 Subjects of the Study

Since CALM extends ATUA, we reuse all the subjects used for evaluating ATUA, except those that cannot be tested anymore because relying on dismissed server-side APIs.

Table 4.1 shows the selected versions (52 subjects, in total); Table 4.2 provides the number of updated methods for each version, they range from one (version V7 of Activity Diary) to 1430 (Wikipedia's V1), thus being representative of diverse release scenarios (i.e., from bug fixes to major releases). The number of bytecode instructions shows that our subjects vary in complexity (from 3667 to 163303).

4.3.2 Experiment Setup

We compare CALM with ATUA and five SOTA tools: APE[37], TimeMachine [111], Monkey[42], Fastbot2 [46], and Humanoid [56]. APE is the SOTA tool that is more likely to achieve the highest coverage for a one-hour test-budget [44]. Monkey, which employs a pure random testing strategy, is the de-facto standard baseline used in the literature [14, 44]. TimeMachine improves over Monkey by leveraging emulators' snapshots to keep a pool of interesting App states (i.e., reached after improving coverage) to resume testing from, when coverage improvement gets stuck. Fastbot2 is a recent approach reusing App models across versions. Humanoid relies on deep learning to effectively exercise Apps like humans [56].

We tested our subjects with CALM and competing approaches using a test budget of one hour, which is a common choice in several App testing papers [37, 43, 8, 114].

We executed each tool with each updated version ten times. For CALM, for each of these ten experiments, we simulated a realistic usage scenario by first testing the initial version of the App considering the entire code as updated, thus deriving an initial App model for V0. We then tested the upgraded versions by reusing the App model generated for the previous App version considered in the same experiment. In total, the experiment took 6940 hours of computing time.

4.3.3 RQ1 - Effectiveness for a Given Test Budget

Experimental Design

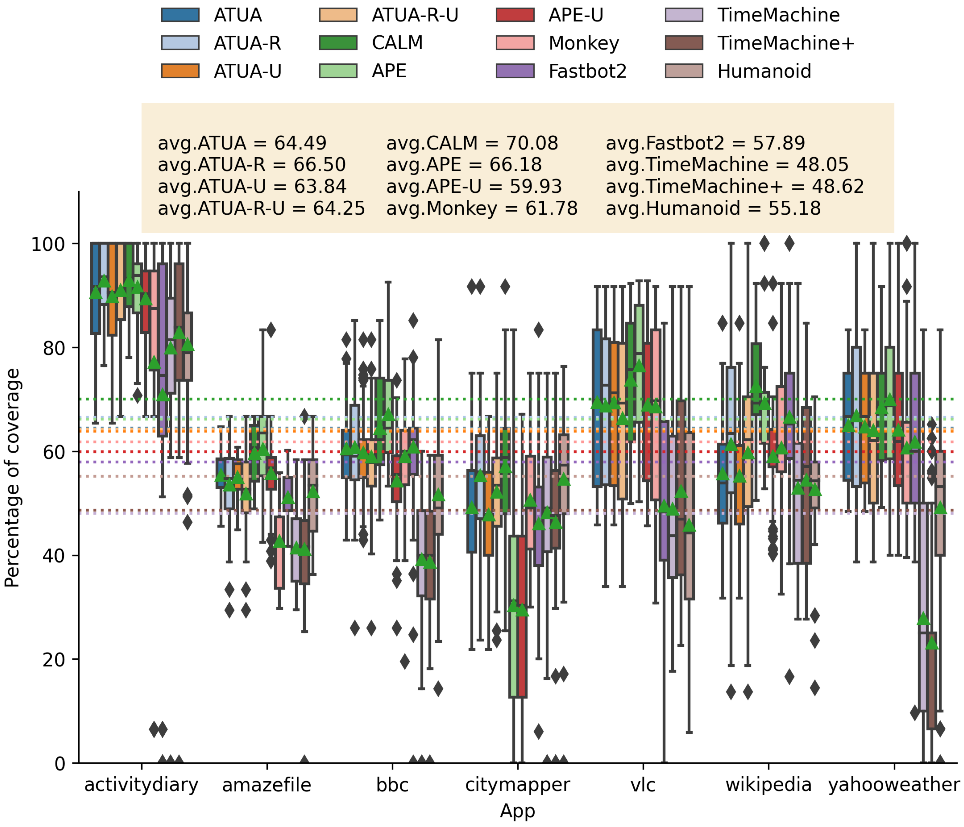
Since we are interested in exercising code that is likely affected by changes (updated methods), we compare CALM with the other approaches in terms of percentage of covered updated methods (hereafter, target method coverage) and instructions belonging to updated methods (hereafter, target instruction coverage).

Since the identification of functional failures can only be based on the visual inspection of App screens rendered after every input action, it is necessary to compare the coverage results obtained when a similar number of App screens is inspected, so that we can assume the effort required to detect failures is similar across competing approaches.

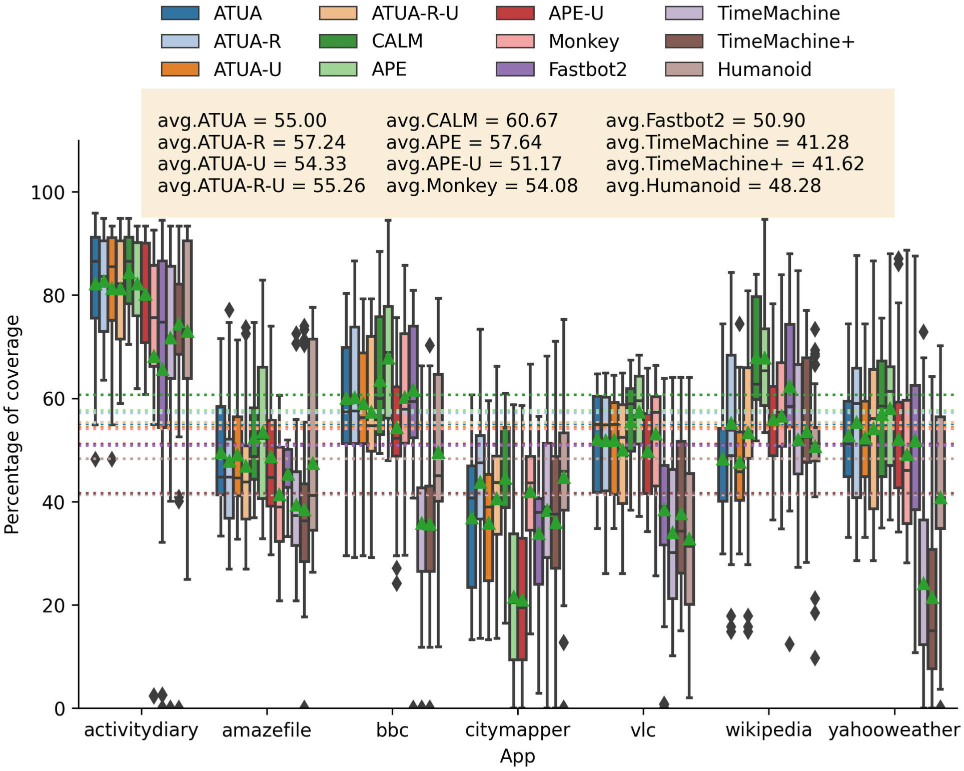
We assume that engineers apply the strategy presented in Section 3.2: they inspect only the App screens generated by useful target actions (UTAs), which are defined as actions contributing to increasing the coverage of instructions belonging to updated methods. In our analysis, we therefore compare the coverage obtained for a same number of UTAs, which enables comparing effectiveness for a same failure detection cost.

During testing, we identify UTAs and the target instructions they cover. Then, for each subject version, we compute the average number N of UTAs generated by CALM, and we select, for each execution of the other testing tools on the same subject, the first N UTAs being triggered. We then compute the target method and instruction coverage achieved with the selected UTAs. We extended ATUA, APE and TimeMachine to collect the instructions covered by each UTA. For Monkey and Fastbot2, it was not possible to implement the same extension; therefore, we report the coverage achieved by these tools with all the inputs triggered in one hour. Additionally, for completeness, we report the target coverage achieved by APE and ATUA with all the inputs triggered in one hour. Please note that, in practice, it would be infeasible for engineers to visually inspect all the App screens rendered with Monkey, APE, and Fastbot2 because of the large number of inputs they trigger [33]; however, Monkey enables us to gain insights about the input space (i.e., how simple it is to exercise target methods without guidance). To perform an ablation study, we implemented a version of ATUA (ATUA-R) that reuses models across versions (i.e., implements CALM's Steps 1 and 2 but not the heuristics of Step 3); also, we implemented a version of TimeMachine (i.e., TimeMachine+) that focuses on target instruction coverage to determine interesting states.

To positively answer RQ1, CALM should achieve a significantly higher target method and instruction coverage than competing approaches, for a same number of UTAs. We determine statistical significance of the difference using a non-parametric Mann Whitney U-test (with α= 0.05). Further, since performance fluctuations across App versions might be expected, we report on the number of versions in which CALM performs better. To this end, we rely on the Vargha and Delaney's A_{12} statistics [100], a non-parametric effect size measure, applied to the ten execution results obtained for a given version. Following standard practice, CALM is deemed to perform better than other approaches when the difference is statistically significant and $A_{12} > 0.56$.



(a) Target method coverage.



(b) Target instruction coverage.

Figure 4.7: Distribution of target method coverage and target instruction coverage.

Table 4.3: Number of versions in which CALM performs significantly better than competing approaches and vice-versa.

Tool	Target method coverage		Target instr. coverage	
	CALM better	CALM worse	CALM better	CALM worse
ATUA-U	41 (78.85%)	0 (0%)	46 (88.46%)	0 (0%)
ATUA-R-U	37 (71.15%)	2 (3.85%)	37 (71.15%)	2 (3.85%)
APE-U	44 (84.62%)	2 (3.85%)	50 (96.15%)	0 (0%)
Monkey	35 (67.31%)	4 (7.69%)	35 (67.31%)	4 (7.69%)
Fastbot2	38 (73.08%)	1 (1.92%)	37 (71.15%)	2 (3.85%)
TimeMachine	48 (92.31%)	0 (0%)	46 (88.46%)	0 (0%)
TimeMachine+	44 (84.62%)	1 (1.92%)	42 (80.77%)	1 (1.92%)
Humanoid	40 (76.92%)	1 (1.92%)	38 (73.08%)	1 (1.92%)

Results

Figures 4.7a and 4.7b show the distributions of the target method and instruction coverage for CALM, for each subject App. The two Figures show similar distributions; a data point is the coverage achieved with one test execution on one App version. **CALM is the approach yielding the best results, on average, with 70.08% and 60.67% target method and instruction coverage**, respectively; differences between CALM and other tools are statistically significant. The second-best result is obtained by APE (66.18% and 57.64%), if we consider all the inputs generated. However, to be realistic, we should rely exclusively on UTAs, which make the performance of APE (i.e., APE-U) drop to 59.93% and 51.17%, approximately a 10% and 9% decrease from CALM, respectively. APE performs better than Fastbot2 (57.89% and 50.90%), which differs from previous results [46], likely because Fastbot2 overfits the specific industrial scenarios for which it was developed.

ATUA-U performs better than APE-U (63.84% and 54.33%), while ATUA-R-U performs slightly better than ATUA-U (64.25% and 55.26%) but differences are not significant, which indicates that model reuse alone provides limited benefits without all the heuristics integrated into CALM (CALM performs significantly better than ATUA-R-U).

CALM performs significantly better than APE-U, ATUA-U, and ATUA-R-U thus showing that model reuse improves the testing of updated Apps but CALM's heuristics are necessary to effectively reuse models (indeed, CALM performs significantly better than ATUA-R-U). Further, the better performance of CALM over Fastbot2 shows that **model reuse alone, without appropriate strategies to drive testing, is not sufficient to effectively test updated methods**.

The need for appropriate testing strategies is also highlighted by the poor performance of Monkey, TimeMachine, and Humanoid. The performance of Humanoid and TimeMachine does not change when either considering all the inputs or UTAs only; for such reason we do not report Humanoid-U and TimeMachine-U in Figure 4.7. TimeMachine is likely negatively affected by the cost of taking execution snapshots. TimeMachine+ is the second-worst approach, thus showing that **focusing on target instructions is not sufficient to test modified functionalities** but a dedicated approach (i.e., CALM) is needed. Humanoid poorly performs likely because it tends to focus on the main App features, which might not be the modified ones, in addition to being affected by other limitations [10]. Please note that both Fastbot2 and Monkey also lead to a much higher number of output screens to be manually verified (i.e., 4645, for Fastbot2, and 51,500, for Monkey, on average for each version versus a range between 1 and 186 for CALM, 35 on average).

Table 4.3 provides the number of App versions in which ATUA performs significantly better than competing approaches and vice-versa. Table 4.3 shows that **CALM performs significantly better than competing approaches for a significantly larger number of versions**, thus showing it is the best choice to incrementally test App versions.

4.3.4 RQ2 - Effectiveness Over Time

Metrics

We study the effectiveness of CALM, for increasing testing time budgets and updates of different magnitude. To measure such magnitude we rely on the proportion of updated methods because it enables us to compare results achieved with Apps of different sizes.

Based on the distribution of the number of App versions per percentage of updated methods in our subjects, we identified three distinct patterns in App development (e.g., from bug fixes to major releases). Tiny updates with [0%,1%) updated App methods are very frequent (52.85% of our versions); small updates with [1%,10%) updated methods are relatively frequent (34.62% of versions); medium updates with [10%,30%) updated methods are much less frequent (11.54% of versions).

As for RQ1, we rely on code coverage as a proxy for effectiveness. We focus on target instruction coverage because method coverage is likely to show high variations between test executions when updates have limited magnitude.

During RQ1 experiments, we traced timestamps and the target instruction coverage for every input action. To address RQ2, we focus on the coverage achieved by each technique, after every minute, considering UTAs only, as in RQ1.

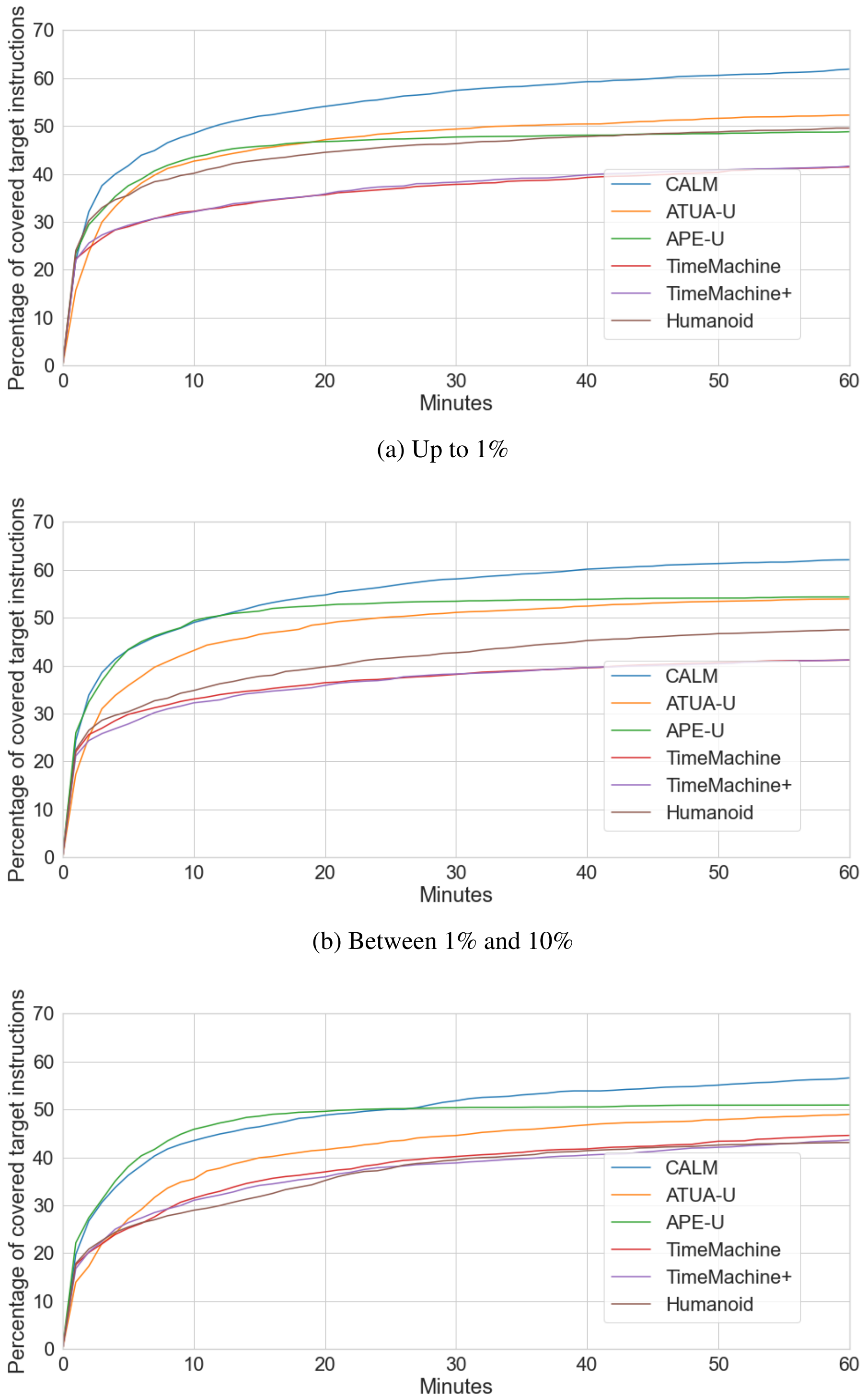
In our analysis, we exclude Monkey and Fastbot2 since they are not practically applicable in our context given that that they do not enable the selection of UTAs.

For each update size range, we compute the average target instruction coverage for all the ten experiment runs of all the App versions having a number of updated methods in that range; we discuss the significance of their difference across ranges based on the Mann Whitney U-test (with $\alpha = 0.05$).

Results

Figure 4.8 depicts the average target instruction coverage over time. Our results show that **CALM always achieves higher average target instruction coverage than ATUA, for any test budget**, which indicates that model reuse, including all the CALM's optimizations, is always the best choice; this would not have been the case if the reused models were driving CALM towards exercising obsolete input sequences leading to unexpected App states. CALM always performs significantly better than Humanoid, TimeMachine, and TimeMachine+. APE-U, however, performs slightly better than CALM in the first minutes of execution (e.g., model loading cost may negatively affect CALM), but then CALM overcomes APE. Interestingly, the difference in performance between the two approaches and the moment in which CALM takes over depends on the magnitude of the change.

With up to 1% updated methods, which is the most frequent case (more than half of our subject versions), APE-U performs significantly better only in the first minute of execution but with a limited improvement of 1.4 percentage points (pp). CALM starts faring significantly better than APE-U after 2 minutes of execution with the average difference between CALM and APE-U increasing from 5 (at 2 minutes of testing) to 13 pp (after 60 minutes).



(a) Up to 1%

(b) Between 1% and 10%

(c) More than 10%

Figure 4.8: Average % of covered target instructions across subjects, grouped by magnitude of changes (updated App methods).

With 1% to 10% updated App methods, APE-U performs significantly better only in the first minute (1.6 pp higher). Between 15 and 32 minutes the difference between the two is not significant but CALM's coverage increases linearly from 1.2 pp to 4.75 pp. After 32 minutes the difference is significant (5 pp higher) and then reaches 7.5 pp at 60 minutes. CALM reaches a max average coverage of 62.1% versus 54.3% for APE-U; further, APE-U reaches a plateau at 30 minutes (in the last 30 minutes of APE execution, its mean coverage increases only by 0.9 pp), while CALM keeps improving its coverage.

When the proportion of updated methods is large (more than 10%), APE-U performs slightly better than CALM in the first 26 minutes but differences are not significant and the improvement is moderate (up to 2.4 pp); however, APE-U reaches a plateau at 23 minutes while CALM keeps improving. After 38 minutes the difference between the two approaches is significant, with CALM performing better by 5.6 pp after 60 minutes. When the magnitude of changes is large, a larger test budget is required to observe a significant difference between CALM and APE-U. Such result is expected since, with a larger proportion of updated methods, it is easier to exercise updated methods regardless of the guidance effectiveness. However, the difference between CALM and APE-U keeps increasing for a larger test budget.

We performed an additional experiment with CALM and APE-U executed for two hours. Results are shown in Figure 4.9[1]; when more than 10% of App methods are updated, after two hours, CALM and APE-U achieve a target instruction coverage of 56.19% and 51.92%, respectively. The difference between the two approaches is significant and increases from 2.56 pp (1-hour budget) to 4.26 pp (2-hour budget).

To summarize, CALM always performs **significantly better than Humanoid, TimeMachine, and TimeMachine+**. Also, **for tiny updates (the majority) CALM performs better than APE-U after 2 minutes of test budget, which is reasonable. For larger updates, the larger budget required by CALM is justified by its coverage not reaching a plateau but steadily improving until becoming significantly higher than that of APE-U.**

[1] Please note that the experiment for a 2-hour budget corresponds to new executions of CALM and APE on all the subject Apps; therefore, since the datapoints are not the same as the ones collected for the 1-hour budget, the curves in Figures 4.8 and 4.9 do not exactly match but show similar trends.

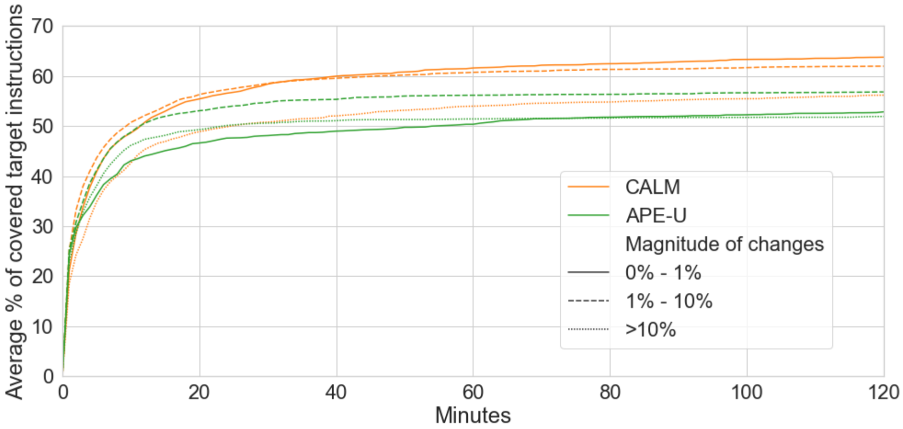


Figure 4.9: Average % of covered target instructions across subjects up to 2 hours, grouped by magnitude of changes (updated App methods).

4.3.5 Threats to Validity

Internal validity. To minimize *implementation errors*, we have carefully tested CALM before running our experiments. For the selected competing state-of-the-art tools, we relied on the versions released by their authors, which had been extensively used in related work.

Conclusion validity. To avoid violating the *assumptions of parametric statistical tests*, we rely on a non-parametric test and effect size measure (i.e., Mann Whitney U-test and the Vargha and Delaney's A_{12} statistics, respectively). To ensure *reliability*, our measurements (i.e., code coverage) have been collected through widely used, open-source tools.

Construct validity. The constructs considered in our work are effectiveness and cost. Effectiveness is measured through two reflective indicators, which are target method coverage and target instruction coverage. We rely on code coverage because it is a common measure of effectiveness for functional testing [13, 107]. Cost is measured in terms of the number of target actions whose effects (i.e., resulting App screens) should be inspected to determine test outcome, as discussed in Section 4.3.3.

External validity. We have considered seven popular Apps, used in related work and in the empirical assessment of ATUA, which enabled fair comparison to discuss the coverage improvements enabled by CALM. For each App, we considered up to ten App versions, based on their availability, for a total of 52 App versions tested. The considered Apps are diverse in terms of features, the overall number of instructions, and updated instructions between versions.

To account for randomness, we tested each App version ten times with every testing tool considered. Despite the high computational cost (6940 test execution hours, in total), this enabled us to derive solid statistical results for the comparison of different tools.

4.4 Conclusion

We presented CALM, a technique to efficiently test App updates by relying on models learned with previous App versions. It relies on static analysis to identify GUI components modified across versions and adapt App models accordingly (e.g., reuse abstract states for renamed Windows); further, it integrates four heuristics addressing the limitations of model inference that are exacerbated in the presence of model reuse: It infers *layout-guarded abstract transitions*, which deal with non-deterministic transitions; it derives *probabilistic Action sequences* to deal with *state explosion*; it detects model states that are new but compatible with previously executed Action sequences (i.e., *backward-equivalent*); it relies on *online and offline model refinement* to identify and remove obsolete states.

Our empirical evaluation shows that CALM leads to a coverage of updated methods and instructions that is higher than the second best SOTA approach by 6 percentage points (pp) for a one-hour test budget. That difference keeps steadily widening as the test budget increases and is larger for smallest updates (13 pp), which are the most frequent.

Chapter 5

App Output Selection Techniques for Test Oracles based on Visual Inspection

5.1 Introduction

In Chapter 3 and Chapter 4, we have presented two approaches for testing updated Apps that aim at exercising all the instructions belonging to new or modified methods while containing testing costs. For our approaches, since test input generation is automated, the main cost component is the test oracle, that is, the manual verification of the outputs of the AUT, as explained below.

Unfortunately, test oracle automation remains an open research problem. In general, only faults leading to crashes can easily be detected by test generation techniques (e.g., through log inspection); however, they constitute a limited proportion of the faults affecting Apps (e.g., 31% of faults affecting Android Apps, according to a recent study [31]). Although approaches supporting engineers in detecting non-crashing failuresexist [26, 29, 27, 23, 24, 74, 25, 22, 30, 115], a recent study [31] has reported that they can target only a limited portion of the functional faults affecting real-world Apps[1]. Furthermore, less than 10% of such faults could actually be detected by existing tools.

Because of the limitations of existing test oracle automation techniques, visual inspection remains necessary. For example, as suggested in previous chapters, when an AUT is exercised using a test automation tool, output verification can be conducted by visually inspecting the screenshots recorded before and after a test action (e.g., clicking on a button). However, since test automation tools exercise Apps with a large number of test actions, to limit testing costs, it is necessary to define strategies to select a subset of the actions that are likely to include erroneous outputs. For ATUA and CALM, assessments of cost-effectiveness have been conducted by selecting all the screenshots generated by those actions that contribute to increasing the coverage of target methods (i.e., new or modified methods). However, it remains to be investigated if more effective action selection strategies can be defined.

A necessary condition for a target method to be correct is that the App output screen rendered after its execution is not erroneous. Therefore, as suggested in previous chapters, test engineers should at least verify the screenshots recorded after the execution of actions exercising target methods (hereafter, *target actions*). Consequently, instead of defining strategies that potentially select any screenshot captured during testing, in this chapter, we propose a number of strategies that aim at selecting a minimized set of target actions (hereafter, *MTA*) whose outputs should be visually inspected by engineers. Our strategies are based on the analysis of both code coverage and action effects (i.e., how an action changes the state of the AUT), two aspects already considered in related work (e.g., differential testing of Apps running on different smartphones [26]).

We conducted an empirical study with real faults in Android Apps with two objectives (1) assess the proportion of functional faults that can be detected by inspecting the screenshots generated by target actions and (2) identify the strategy leading to the best balance between cost (i.e., minimizing the number of actions whose screenshots should be inspected) and effectiveness (i.e., maximizing the number of faults being detected). Since our strategies for the selection of MTA (hereafter, *MTA strategies*) are effective only if the target actions triggered by a test generation technique lead to failures, we also evaluate how effective

[1] 84 faults out of the 399 faults (21%) inspected by Xiong et al., as reported in Table 5 of their work [31].

CALM is at exposing functional failures. We rely on CALM because it has been shown to outperform ATUA and other approaches.

Our results show that (1) the majority (81%) of faults[2] can be detected by inspecting the screenshots associated to target actions, thus corroborating the choices we made when defining our MTA strategies; (2) CALM generates actions that enable the detection (i.e., show failures) of 72% (8/11) of the faults and can achieve 91% (10/11) when it is used to test one method at a time; (3) our MTA strategies could reduce test oracle costs by 52% to 84% while achieving a fault detection rate from 75.2% to 100%. Precisely, engineers may choose between (1) a strategy that minimizes costs (i.e., a cost saving strategy) and helps detecting 68.9% of the faults, on average, in our experiments and (2) a strategy that maximizes the number of faults detected (100%, in our experiments) but comes at a higher cost (almost 7 times the other strategy). Nevertheless, there is still a cost saving of 42% if the latter strategy is adopted.

This Chapter proceeds as follows. Section 5.2 provides definitions for the terminology used in the Chapter. Section 5.3 describes our MTA strategies. Section 5.4 reports the results of our empirical study. Section 5.5 provides final remarks.

5.2 Definitions

Target action A *target action* is an action, among the ones executed during testing, exercising at least one target method (i.e., a method introduced in the new App version or a method modified in the new app version). We refer to the set of all target actions triggered during test execution as the *full set of target actions*. Target actions are a subset of all the actions executed during testing; they can be identified through dynamic program analysis. For example, ATUA and CALM identify target actions by tracing the code exercised after triggering each test action and by selecting the actions that exercise a target method (see Chapters 3 and 4). Precisely, they rely on an extension of the coverage instrumentation component of Droidmate-2 (see Section 3.3) that monitors the beginning and end of an event handler executed after a test input and reports all the executable instructions exercised

[2] We use the term fault because, by selecting App outputs that are likely to be erroneous, our approach enables detecting the presence of faults in the App under test. Our approach selects outputs that are erroneous; in other words, our approach shows failures that enable the detection of faults.

in-between.

Minimized sets of Target Actions To reduce the human effort, it is necessary to reduce the number of target actions whose screenshots (i.e., the screenshots recorded before and after triggering an action) should be visually inspected. We refer to such reduced set of target actions as *"Minimized set of Target Actions"* (shortly, *MTA*). Essentially, an MTA should consists of target actions that enable testing all the input partitions[3] for the functionalities that have been modified or introduced in a new App version.

5.3 Strategies for the Generation of Minimized Sets of Target Actions

In this Section, we present our strategies for the selection of MTA (hereafter, *MTA strategies*). To select MTA, we rely on two types of data that can be collected during test execution: code coverage and action effects.

Table 5.1 provides the full list of proposed strategies. They can be grouped into three categories: *coverage-based strategies*, *action-effect-based strategies*, and *combined strategies*; they are described below.

5.3.1 Coverage-based Strategies

Our coverage-based strategies result from the combination of three coverage criteria with two different approaches for action selection (*chronological approach* and *greedy approach*). Below, we introduce some supporting notation used to define our coverage criteria, and describe the two approaches along with the resulting strategies.

Coverage Notation

Before introducing our coverage-based strategies, we introduce some supporting notation.

[3] In software testing, an input partition is a region of the input domain with values that are equivalent from a testing perspective (i.e., they all lead to failures or passing tests) [116].

```
0     double area(int x1, int y1, int x2, int
          y2) {
1         if (x1 > x2)
2             l_x = x1 - x2;
3         else
4             l_x = x1 - x2; // bug
5         if (y1 > y2)
6             l_y = y1 - y2;
7         else
8             l_y = y1 - y2; // bug
9         return l_x * l_y
10    }
```

	a_1	a_2	a_3	a_4
s_1	●	●	●	●
s_2	●		●	
s_3		●		●
s_4		●		●
s_5	●	●	●	●
s_6	●			●
s_7		●	●	
s_8		●	●	
s_9	●	●	●	●

Figure 5.1: Different possibility of statement coverage for a function. On the right, four actions a_1, a_2, a_3, a_4 cover different set of statements (i.e., $s_1, s_2, \ldots, s_9$) corresponding to line number of the function *area* on the left.

- Σ_F: the full set of target actions triggered during a test session (e.g., the execution of CALM for one hour on an App version).

- Σ_{MTA}: the MTA for the test session under analysis.

- C_a: the set of instructions covered by an action a. For example, if we consider the function *area* in Figure 5.1, an action a_1 may exercise the instructions corresponding to lines 1, 2, 5, 6, and 9 while the action a_2 may exercise instructions 1, 3, 4, 5, 7, 8, and 9. More formally, for all the actions in Figure 5.1, we obtain the following instruction sets:

$$C_{a_1} = \{s_1, s_2, s_5, s_6, s_9\}$$

$$C_{a_2} = \{s_1, s_3, s_4, s_5, s_7, s_8, s_9\}$$

$$C_{a_3} = \{s_1, s_2, s_5, s_7, s_8, s_9\}$$

$$C_{a_4} = \{s_1, s_3, s_4, s_5, s_6, s_9\}$$

- C_Σ: the set of instructions covered by the set of actions Σ; it is the union of instructions covered by actions belonging to Σ.

- F_Σ: the instruction footprint of a set of actions Σ. The instruction footprint of a set of actions is a set of sets; precisely, it is a set including, for each action, the set of instructions covered by it. Based on the example in Figure 5.1, we can define the instruction footprint of the sets of actions $\{a_1, a_2\}$ and $\{a_1, a_2, a_3\}$ as follows:

$$F_{\{a_1,a_2\}} = \big\{\{s_1, s_2, s_5, s_6, s_9\}, \{s_1, s_3, s_4, s_5, s_7, s_8, s_9\}\big\}$$

$$F_{\{a_1,a_2,a_3\}} = \big\{\{s_1, s_2, s_5, s_6, s_9\}, \{s_1, s_3, s_4, s_5, s_7, s_8, s_9\}, \{s_1, s_2, s_5, s_7, s_8, s_9\}\big\}$$

Although, by looking at code coverage, the two sets of instructions are equivalent (i.e., $C_{\{a_1,a_2\}} = C_{\{a_1,a_2,a_3\}}$), their coverage footprints are not. This happens because the coverage footprint helps determine that the paths exercised by the two sets of actions are different. From a testing perspective, the set of actions $\{a_1, a_2, a_3\}$ is better than the set $\{a_1, a_2\}$ as it detects the fault. Indeed, the example in Figure 5.1 is affected by a fault that leads to failures (i.e., a negative area) only if specific execution paths are exercised: if both `else` instructions are executed, the two faulty lines lead to two negative numbers, thus ending with a positive number (i.e., a valid result).

- Asterisk $*$: An asterisk is added to the C and F notations above to indicate that, instead of considering all the instructions, we consider only the instructions belonging to a target method (hereafter, target instructions). Consequently, three more notations are defined:
  - C^*_a the set of target instructions exercised by an action a;
  - C^*_Σ the set of target instructions covered by the set of actions Σ;
  - F^*_Σ the target instruction footprint of the set of actions Σ.

Coverage Criteria

To derive an MTA based on code coverage, we propose three criteria: *target instruction coverage*, *target instruction footprint coverage*, and *instruction footprint coverage*.

Target instruction coverage ensures that the MTA includes a number of actions that maximize the number of instructions that belong to a target method and are exercised. An MTA satisfies the target instruction coverage criterion if every target instruction covered by the full set of target actions is also covered by the MTA. More formally,

$$s \in C^*_{\Sigma_F} \implies s \in C^*_{\Sigma_{MTA}}$$

Target instruction footprint coverage ensures that the actions in the MTA exercise all the footprints belonging to the full set of target actions. More formally,

$$f \in F^*_{\Sigma_F} \implies f \in F^*_{\Sigma_{MTA}}$$

Our rationale is that the instruction footprint exercises paths and, therefore, data dependencies not exercised otherwise. This is the case of the example of Figure 5.1, as explained above.

Instruction footprint coverage extends the identification of instruction footprints to the whole set of App instructions, to exercise inter-class dependencies.

More formally,

$$f \in F_{\Sigma_F} \implies f \in F_{\Sigma_{MTA}}$$

Coverage Approaches

The three criteria above lead to different results along with the approach adopted for the selection of actions. We propose two approaches: the chronological approach and the greedy approach.

Algorithm 2 Algorithm to construct an MTA with the chronological approach

```
Input:
    $T = \{a_0, a_1, \dots, a_n\}$      ▷ full set of target actions in ascending chronological order
    $p$      ▷ property that an action needs to satisfy to be included in MTA set.
Output: $U = \{a_0, a_1, \dots, a_m\} | U \subseteq T$      ▷ an MTA set.
1: $U \leftarrow \emptyset$
2: for all $a \in T$ do
3:     if $a \vdash p$ then      ▷ $a$ satisfies $p$
4:         $U = U \cup \{a\}$
   return $U$
```

Chronological approach The *chronological approach* follows the chronological order of actions in the test input sequences produced by a test automation tool (e.g., the sequence of action in CALM's and ATUA's GSTG) and selects an action if it satisfies a given property). It is captured by Algorithm 2, which traverses each action in the full set of target actions T (Line 2) and selects an action if it satisfies a given property p (Lines 3-4).

We can combine the chronological approach with the coverage criteria into three properties:

- Increasing target instruction coverage: an action a satisfies this property if it exercises at least one target instruction not exercised by previously selected actions.

$$p_{tic} \models C^*_a \nsubseteq C^*_{\Sigma_{MTA}}$$

- Increasing target instruction footprints: an action satisfies this property if it exercises a set of target instructions not matching any set of target instructions exercised by the previously selected actions; in other words, the action leads to a target instruction footprint that differs from the target instruction footprint obtained with the already selected actions.

$$p_{tif} \models C^*_a \notin F^*_{\Sigma_{MTA}}$$

- Increasing instruction footprints: an action satisfies this property if it exercises a set of instructions not matching any set of instructions exercised by the previously selected actions.

$$p_{if} \models C_a \notin F_{\Sigma_{MTA}}$$

The first three rows of Table 5.1 describe three strategies relying on the three properties proposed above.

Algorithm 3 Algorithm to construct an MTA using a greedy approach

```
Input:
    $T = \{a_0, a_1, \ldots, a_n\}$                    ▷ full set of target actions
    $p$          ▷ property that an action needs to satisfy to be included in MTA set.
Output: $U = \{a_0, a_1, \ldots, a_m\} | U \subseteq T$             ▷ an MTA set.
    $U \leftarrow \emptyset$
    while $T \neq \emptyset$ do
        $C_d \leftarrow \{a \in T, a \vdash p\}$        ▷ $C_d$: set of candidate actions (i.e., satisfying $p$)
        if $C_d \neq \emptyset$ then
            $sel \leftarrow \arg\max_{a \in C_d} \omega(a, U)$      ▷ $\omega$ : fitness function for greedy prioritization
            $U = U \cup \{sel\}$
            $T = T \setminus \{sel\}$
        else
            break
    return $U$
```

Greedy approach The *greedy approach* iteratively selects, from the subset of generated actions that satisfy a given property p, the action maximizing a given fitness score. It is captured by Algorithm 3, which differs from Algorithm 2 because, instead of selecting the first action satisfying p, it iteratively identifies, as candidates, all the actions in the set T satisfying P (Line 3) and selects the action maximizing a fitness score (lines 5-6). By selecting, at each iteration, the action that maximizes a given fitness score, we aim to minimize the number of selected actions; however, there is no guarantee that the algorithm identifies the minimal set of actions, we leave such optimization (e.g., through meta-heuristic search or linear programming) to future work. We rely on the same properties defined for the chronological approach. The fitness score of each action is computed by a dedicated fitness function ω; we provide two alternative definitions for ω that maximize two metrics of interest:

- *Number of target instructions*: we maximize the number of target instructions exercised by an action but not by other actions already selected for the MTA.

$$\omega_{ti}(a, \Sigma_{MTA}) = |C^*_a \setminus C^*_{\Sigma_{MTA}}|$$

- *Number of instructions*: we maximize the number of instructions exercised by an action but not by other actions already selected for the MTA.

$$\omega_i(a, \Sigma_{MTA})) = |C_a \setminus C_{\Sigma_{MTA}})|$$

In Table 5.1, the strategies CC_WTI_PTIC, CC_WI_PTIC, and CC_WI_PTIF rely on the greedy approach. They result from combining the fitness functions $\omega_{ti}$and $\omega_i$with the properties $p_{tic}$and p_{tif}. We do not combine the fitness functions $\omega_{ti}$and $\omega_i$with p_{if}because, by considering all the instruction footprints (i.e., not only the target instruction footprints), p_{if}leads to selecting most of the actions and thus is very expensive and unlikely to lead to a reduced set of actions when used with a greedy approach. Also, we avoid combining $\omega_{ti}$with p_{tif}because it would lead to the same results as CC_WTI_PTIC; indeed, the subset of actions exercising a target instruction footprint not already selected (property p_{tif}) and increasing the coverage of target instructions (fitness ω_{ti}) is the same as what is achieved by

CC_WTI_PTIC, that is, the subset of actions increasing target instruction coverage (property p_{tic}) and increasing the coverage of target instructions (fitness ω_{ti}).

5.3.2 Action-effect-based Strategies

We use the term *action effect* to indicate the effect of an action on the state of the App. For GUI-driven Apps, an App state can often be characterized in terms of its GUI tree or a screenshot of the App being rendered on the screen. Also, in several App testing approaches, including ATUA and CALM, GUI states are captured by abstraction functions, and related work has shown that action effects can be effectively measured as the difference between the App state before and after performing an Action [22, 30]. The difference between two App states can thus be computed as the minimal tree edit distance between GUI trees, abstract states, or differences between two screenshots.

Recent empirical studies have shown that relying on GUI tree difference to determine if the state of an App changes after a sequence of actions may result into a high number of false negatives compared to screenshot distance [117]. Indeed, GUI trees are incapable of capturing some visual properties such as color. For example, in Figure 5.2, a data loss fault in the EditActivity Window from the Activity Diary App causes the "Activity color" to revert to its original color after a rotation. Unfortunately, GUI trees do not capture such visual information.

Because of the above, we rely on screenshot distances to measure action effects. Precisely, we rely on two state-of-the-art feature-based image distance metrics, which are SIFT (Scale-Invariant Feature Transform) [118] and SSIM (Structural Similarity Index) [119]. Hereafter, for simplicity, we refer to such distance metrics with the generic name of Action-effect metrics; each Action is associated to one Action-effect metric value (*Action-effect value* for short), which is computed by relying on the selected metric.

Algorithm 4 details how we select an MTA with our action-effect strategy. It receives as input the full set of target actions T, an abstraction function $\mathcal{L}_1$, an image distance function d used to compute the action-effect value, and an histogram distance function hid. First, we group actions by relying on the state of the Widget triggered by the action (Line 1); precisely, since actions exercising different App Windows should naturally belong to different groups,

Algorithm 4 Algorithm of constructing an MTA set with screenshot-based strategy

```
Input:
    $\Sigma_F = \{a_0, a_1, \dots, a_n\}$: full set of target actions
    $\mathcal{L}_1$: abstraction function used to group target actions
    $d(a)$: image distance function, returns the distance between the source screenshot and the destination screenshot of action $a$    ▷ SIFT, or SSIM
    $hid(a_1, a_2)$: histogram distance function, returns the distance between the histogram of the source screenshot of action $a_1$ and one of action $a_2$    ▷ GrayScaleHistogramCorrelation
Output: $\Sigma_{MTA} = \{a_0, a_1, \dots, a_m\} | \Sigma_{MTA} \subseteq \Sigma_F$: a Minimized set of target action.
$G_{\Sigma_F} \leftarrow$ group target actions by relying on $\mathcal{L}_1$, it returns a group of action sets $\{A_0, A_1, \dots, A_n\}$
$\Sigma_{MTA} \leftarrow \emptyset$
for all $A \in G$ do
    $sel_{\text{argmax}} \leftarrow \arg\max_{a \in A} d(a)$
    $sel_{\text{argmin}} \leftarrow \arg\min_{a \in A} d(a)$
    $sel_{\text{argmedian}} \leftarrow \arg\operatorname{median}_{a \in A} d(a)$
    $\Sigma_{MTA} \leftarrow \Sigma_{MTA} \cup \{sel_{\text{argmax}}, sel_{\text{argmin}}, sel_{\text{argmedian}}\}$
    $A \leftarrow A \setminus \Sigma_{MTA}$
    if $A \neq \emptyset$ then
        $sel \leftarrow \arg\max_{a \in A}(average(D^a = \{hid(a, a'), \forall a' \in A \land a' \neq a\})$
        $\Sigma_{MTA} \leftarrow \Sigma_{MTA} \cup \{sel\}$
return $\Sigma_{MTA}$
```

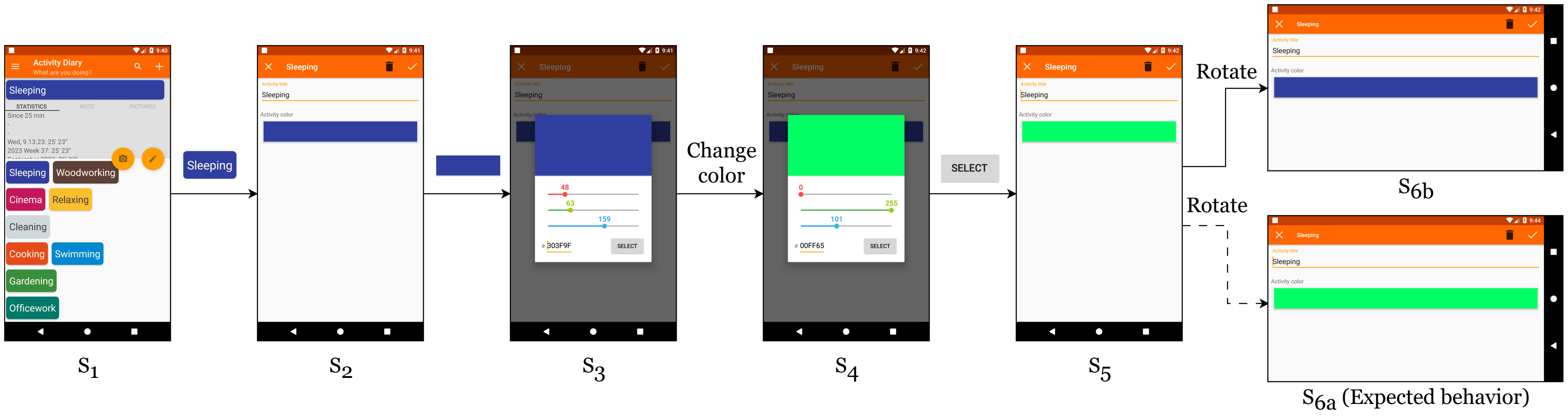


Figure 5.2: A functional fault from the ActivityDiary app. The recently changed color is supposed to be kept after the rotation (i.e., $S_5 \rightarrow S_{6a}$) but it is reverted to the one before the change (i.e., $S_5 \rightarrow S_{6b}$)

(1) we first identify the actions exercising a same App Window, (2) we apply ATUA's $\mathcal{L}_1$ GUI abstraction function to their departing state, and, finally, (3) we group together actions targeting a Widget having a same abstract representation (i.e., a same AVM). Then, since our objective is to select a subset of actions, for each group of actions (Line 3), we select a subset of actions that represents the group's behaviour. Precisely, for each action group, we select:

- The action with the *highest action-effect value* in the action group (Line 4); it is the action leading to the most noticeable state change and thus of interest from a testing perspective (i.e., it may alter the GUI in an anomalous way).

- The action resulting in the *smallest action-effect value* (Line 5); it is the action leading to the least noticeable change in the GUI (potentially, no change) and may thus indicate an anomalous output (e.g., the GUI does not change at all, which might be due to a fault).

- The action with a *median action-effect value* (Line 6); it is an action that should be representative of a common use of the App and thus be inspected.

- If more actions are available (Line 9), an action that *departs from a state that is uncommon* (Line 10). Such state is determined by applying histogram distance (i.e., $hid(i_1, i_2)$) to all the pairs of screenshots taken before executing any action, and identifying the screenshot with the largest average distance from all the other screenshots. It should enable detecting actions that depart from rare states, which might, therefore, not have been foreseen by App developers. To compute such distance, we rely on Grayscale Histogram Correlation (GHC [120]), which is a common choice for image comparison because it enables the comparison of images based on their color profiles [121]. In a preliminary investigation with one of our case studies (Amaze File Manager), we observed that App states could be distinguished by focusing on their color histograms; for example, a page with a file list in which all the items are selected would be filled mostly by gray and thus correctly present a large GHC distance from a page with no file selected. Further, histograms are less affected by small changes in the image, which is an ideal choice in our context because we aim at identifying uncommon states (i.e., states with major differences from others).

In Table 5.1, the strategies whose name starts with *AF* are based on action effect; two strategies rely on SIFT as distance function, two rely on SSIM. Further, these two strategy sets each include a strategy relying on the whole Algorithm 4 (the name of these strategies ends with *3M*), and one strategy relying on Algorithm 4 but executed without Line 6 (i.e., without the instructions selecting Actions with a median Action-effect value—their name ends with *2M*).

Combined Strategies

Although strategies based on action-effect account for the semantics of the App (e.g., noticing changes concerning the App GUI although they are not captured by code coverage), since they do not take into account code coverage, they may ignore actions that exercise potentially faulty code. For this reason, we rely on combined strategies that complement MTA derived using action-effect strategies with unselected actions satisfying coverage properties. Precisely, we first employ Algorithm 4 configured for a specific action-effect-based strategy to construct an MTA and then we complement the set by relying on Algorithm 2 configured for a specific coverage-based strategy, but with a set U that, instead of being empty, contains the set generated by Algorithm 4.

In Table 5.1, the strategies whose name starts with *CB* are combined strategies. All of them combine the four Action-effect-based strategies with coverage-based strategies. We identify four groups of strategies: the ones combining Action-effect strategies with CC_WTI_PTIC and CC_WI_PTIC, the ones combining Action-effect strategies with CC_PTIF, and the ones combining Action-effect strategies with CC_PIF. Table 5.2 provides the mapping between the MTA strategies considered in this work and the properties described earlier in this Section.

5.4 Empirical Evaluation

We aim to answer the following research questions.

RQ1. *Do target action screenshots enable detecting functional faults through visual inspec-*

Table 5.1: List of proposed MTA set selection strategies

UTA strategy name	**Description**
CC_PTIC	Chronologically select target actions increasing target instruction coverage (relies on property p_{tic})
CC_PTIF	Chronologically select target actions covering distinct sets of instructions (i.e., leading to a footprint different from the one of the already selected actions, relies on property p_{tif}).
CC_PIF	Chronologically select target actions covering distinct sets of instructions (relies on property p_{if}).
CC_WTI_PTIC	Greedily select target actions increasing target instruction coverage (property p_{tic}) and maximizing the coverage of target instructions (fitness ω_{ti}); it enables selecting the Actions that maximize target instruction coverage.
CC_WI_PTIC	Greedily select target actions increasing target instruction coverage (property p_1) and maximizing the coverage of instructions (fitness ω_i); it enables selecting, among the Actions that increase target instruction coverage, the ones that maximize the coverage of the whole App.
CC_WI_PTIF	Greedily select target actions covering distinct target instruction footprints (property p_2 and maximizing the coverage of instructions (fitness ω_i); it enables selecting, among all the Actions that show a different coverage of target instructions, the ones that maximize the coverage of the whole App differently.
AF_SIFT_2M	Action-effect-based strategy adopting SIFT as image distance function but selecting only actions yielding the largest distance and the smallest distance (lines 4-5 in Algorithm 4).
AF_SIFT_3M	Action-effect-based strategy adopting SIFT as image distance function, corresponding exactly to Algorithm 4.
AF_SSIM_2M	Action-effect-based strategy adopting SSIM as image distance function but selecting only actions yielding the largest distance and the smallest distance (lines 4-5 in Algorithm 4).
AF_SSIM_3M	Action-effect-based strategy adopting SSIM as image distance function, corresponding exactly to Algorithm 4.
BC_SIFT_2M_WTI_PTIC BC_SIFT_3M_WTI_PTIC BC_SSIM_2M_WTI_PTIC BC_SSIM_3M_WTI_PTIC	These four combined strategies first employ AF_SIFT_2M, AF_SIFT_3M, AF_SSIM_2M, and AF_SSIM_3M respectively, and then select additional target actions increasing the target instruction coverage (property p_{tic}) and maximizing the coverage of target instructions (fitness ω_{ti}).
BC_SIFT_2M_WI_PTIC BC_SIFT_3M_WI_PTIC BC_SSIM_2M_WI_PTIC BC_SSIM_3M_WI_PTIC	These four combined strategies first employ AF_SIFT_2M, AF_SIFT_3M, AF_SSIM_2M, and AF_SSIM_3M, respectively, and then select additional target actions increasing target instruction coverage (property p_{tic}) and maximizing the coverage of instructions (fitness ω_i).
BC_SIFT_2M_WI_PTIF BC_SIFT_3M_WI_PTIF BC_SSIM_2M_WI_PTIF BC_SSIM_2M_WI_PTIF	These four combined strategies first employ AF_SIFT_2M, AF_SIFT_3M, AF_SSIM_2M, and AF_SSIM_3M, respectively, and then select additional target actions increasing target instruction footprints in chronological order (property p_{tif}).
BC_SIFT_2M_PIF BC_SIFT_3M_PIF BC_SSIM_2M_PIF BC_SSIM_3M_PIF	These four combined strategies first employ AF_SIFT_2M, AF_SIFT_3M, AF_SSIM_2M, and AF_SSIM_3M, respectively, and then select additional target actions increasing instruction footprints in chronological order (property p_{if})

Table 5.2: Mapping of MTA strategies with parameters used in Algorithms 2, 3 and 4.

UTA strategy name	**Category**	p_{tic}	p_{tif}	p_{if}	$\omega_{\mathbf{ti}}$	$\omega_{\mathbf{i}}$	**SIFT**	**SSIM**	**2M**	**3M**
CC_PTIC	CC	✓					-	-	-	-
CC_WTI_PTIC	CC	✓			✓		-	-	-	-
CC_WI_PTIC	CC	✓				✓	-	-	-	-
CC_PTIF	CC		✓				-	-	-	-
CC_WI_PTIF	CC			✓		✓	-	-	-	-
CC_PIF	CC			✓			-	-	-	-
AF_SIFT_2M	AF	-	-	-	-	-	✓		✓	
AF_SIFT_3M	AF	-	-	-	-	-	✓			✓
AF_SSIM_2M	AF	-	-	-	-	-		✓	✓	
AF_SSIM_3M	AF	-	-	-	-	-		✓		✓
BC_SIFT_2M_WTI_PTIC	CB	✓			✓		✓		✓	
BC_SIFT_3M_WTI_PTIC	CB	✓			✓		✓			✓
BC_SSIM_2M_WTI_PTIC	CB	✓			✓			✓	✓	
BC_SSIM_3M_WTI_PTIC	CB	✓			✓			✓		✓
BC_SIFT_2M_WI_PTIC	CB	✓				✓	✓		✓	
BC_SIFT_3M_WI_PTIC	CB	✓				✓	✓			✓
BC_SSIM_2M_WI_PTIC	CB	✓				✓		✓	✓	
BC_SSIM_3M_WI_PTIC	CB	✓				✓		✓		✓
BC_SIFT_2M_WI_PTIF	CB		✓			✓	✓		✓	
BC_SIFT_3M_WI_PTIF	CB		✓			✓	✓			✓
BC_SSIM_2M_WI_PTIF	CB		✓			✓		✓	✓	
BC_SSIM_2M_WI_PTIF	CB		✓			✓		✓		✓
BC_SIFT_2M_PIF	CB			✓		✓	✓		✓	
BC_SIFT_3M_PIF	CB			✓		✓	✓			✓
BC_SSIM_2M_PIF	CB			✓		✓		✓	✓	
BC_SSIM_3M_PIF	CB			✓		✓		✓		✓

Legend: CC: Code-Coverage based, AF: Action-Effect based, CB: Combined, ✓: adopted, - : not applicable.

tion? Since the proposed MTA strategies are useful if the screenshots taken before and after the selected actions enable determining the presence of a failure, we aim to measure the proportion of faults that can be identified by such an approach in order to determine the potential impact of the proposed strategies.

RQ2. *How effective is CALM in exposing functional faults?* We aim to report on the proportion of faults that can be effectively exercised by CALM and lead to a failure (i.e., lead to an erroneous output on the App screen). Although the proposed MTA strategies can work with any App testing approach that records the code coverage of Actions and captures screenshots before and after actions, since CALM is the state-of-the-art tool for testing App upgrades, assessing the effectiveness of CALM is required to compute the proportion of faults that can be semi-automatically detected by combining CALM and the proposed strategies.

RQ3. *What is the most cost-effective MTA strategy?* We aim to assess the proposed MTA selection strategies in terms of effectiveness (i.e., likelihood of exposing a fault) and cost (i.e., reduction of the manual effort required to inspect the selected actions and screenshots).

5.4.1 Subjects of the study

To study functional faults in Apps and how the proposed strategies may help minimize the set of outputs to be inspected, it is necessary to consider a set of faults affecting real Apps and have access to all the information necessary to detect and replicate the failures caused by such faults.

Since CALM is the state-of-the-art approach for testing App upgrades, we rely on it to automatically derive test inputs, test outputs, and the information about App states (e.g., screenshots captured before and after triggering an action) required by the proposed MTA strategies. Consequently, for our study, we select functional faults affecting Apps tested with CALM in previous empirical assessments. Further, we focus on open-source Apps because their code repositories come with issue trackers providing means for acquiring relevant fault information (e.g., the location of faults, failure descriptions, information about patches). The shortlisted Apps are ActivityDiary [122] and AmazeFileManager [123], which are the only open-source Apps tested by CALM presenting faults described in issue trackers.

For the selected Apps, we identified all the bug reports concerning faults leading to functional failures noticeable by inspecting the App screen. Also, we considered faults that had been fixed, because it simplifies their understanding (e.g., we can look at the patches to determine what are the faulty lines of code). Last, we selected faults that we could reproduce by executing the faulty App version. We ended up with 11 faults.

To assess the selected faults with CALM, they should have been introduced when implementing or modifying a method for a new App version. However, some of the selected faults had been introduced in App versions that cannot be tested with CALM because they are too old. Therefore, to enable testing all the selected faults with CALM, since each fault consists of a set of erroneous instructions in a method (hereafter, *faulty method*) we proceeded as follows:

- If the faulty method is among the target methods for an App version previously tested with CALM, we select such version as the target for our experiments.

- If an App version tested with CALM includes the faulty method but such method is not among the target methods for that App version (i.e., the fault was introduced previously), we force CALM to test such faulty method by introducing a change in it (i.e., we introduce a new logging instruction).

- If the faulty method does not belong to any App version previously tested with CALM, we identify an App version where it is possible to reintroduce the fault.

Table 5.3 shows the details of the 11 faults selected for our study; we also indicate if the fault was already present in the selected App version or if it has been reintroduced. At a high level, four faults lead to a failure that consists of a UI display issue (e.g., a display of incorrect information), and seven faults lead to a UI interaction problem (e.g., a non-response for an input).

5.4.2 RQ1. Do target action screenshots enable detecting functional faults through visual inspection?

Experiment design We aim to determine what and how many screenshots need to be inspected in order to detect functional faults that manifest on the App screen. To this end, we manually exercised each fault affecting our subjects and kept track of how many and what screenshots should be inspected in order to detect a failure (i.e., determine that the output is incorrect, given previous App states). Our objective is to identify *inspection patterns* characterizing what screenshots should be inspected by engineers to detect a failure. Among all the possible inspection patterns, we aim to report on the proportion of failures detectable with the pattern enabled by our MTA strategies (i.e., looking at the screenshots taken before and after an action that exercises the faulty method). Other patterns may help steer future work (e.g., defining new MTA strategies).

Results From the 11 faults selected for our study, we derived three possible inspection patterns (i.e., ways of inspecting screenshots related to the faulty code to detect failures):

Pattern-1 Inspect the screenshots taken before and after the action that exercises the faulty

Table 5.3: Functional bugs considered in the preliminary evaluation

FaultId	App	Fault description	Github issue ID	Test version	Is reintroduced	Number of target methods
AC01	AD	When the activity edit page is filled, if the screen is rotated then the filled content is lost.	#53	v134	✓	49
AC02	AD	After clicking on the "Delete" button on an activity's edit page, the activity is still present in other pages.	#59	v111	✓	18
AC03	AD	When clicking on a picture of an activity, nothing happens while it is supposed to open the picture with an image viewer.	#162	v134	✓	49
AC04	AD	When undoing the recent activation of an activity, the state of the activated activity is incorrect if there is no activated activity before the undone one.	#286	v134		49
AM01	AM	After manually selecting all file items, if the options menu is opened, the "Deselect All" menu item is not present.	#996	v3.4.1	✓	651
AM02	AM	Triggering the "Select All" menu item when all file items are selected causes all items to be deselected.	#953	v3.4.1	✓	651
AM03	AM	When entering a file/folder name starting with a dot, the dialogue does not warn the user that the file/folder will be hidden.	#1235	v3.4.1	✓	651
AM04	AM	The preselected configuration dialogue for the App's color allow multiple choices with radio buttons instead of single-choice.	#1044	v3.4.1	✓	651
AM05	AM	When creating a new file, a file name ending with ".txt" is not allowed.	#1231	v3.4.1	✓	651
AM06	AM	When searching files, hidden files with matching patterns are shown too.	#1467	v3.4.1		651
AM07	AM	When closing the "Hidden Files" dialogue, the file list is not refreshed and, therefore, the updates from the dialogue do not appear.	#1712	v3.4.1		651

Legends. AD: Activity Diary(https://github.com/ramack/ActivityDiary), **AM**: Amaze File manager (https://github.com/TeamAmaze/AmazeFileManager)

Table 5.4: Inspection patterns for our case study subjects.

FaultId	Pattern-1	Pattern-2	Pattern-3
AC01	✓		
AC02	✓		
AC03	✓		
AC04		✓	
AM01	✓		
AM02	✓		
AM03	✓		
AM04	✓		
AM05	✓		
AM06	✓		
AM07			✓

Legends. AD: Activity Diary, **AM**: Amaze File manager

code in the buggy method and leads to a failure. We call such action *failure-inducing action*.

Pattern-2 Inspect, in addition to the screenshots taken before and after the failure-inducing action, also one screenshot taken before one of the actions that preceded the failure-inducing action because such additional screenshot provides relevant information about the state of the App. For example, in the ActivityDiary App, consider the following test input sequence:

①[Select activity "Study"] → ②[Cancel the current activity] → ③[Select activity "Dinner"]→ ④[Undo]

It leads to a failure because the App, instead of visualizing the previous screen (i.e., the result of ② [Cancel the current activity]), visualizes the content produced as a result of ①[Select activity "Study"]. The failure-inducing action is the undo button. However, to detect the failure (i.e., to determine what was expected to be visualized), we need to inspect the screenshot taken before action ③, which preceded the failure-inducing action.

Pattern-3 Inspect, in addition to the screenshots taken before and after the failure-inducing action, one screenshot taken after one of the actions that follow the failure-inducing action. For example, in the Amaze File Manager App, after closing the "Hidden Files" dialog, the App should automatically perform a refresh to update the displayed file list. However, the constructor method for the dialog contains faulty code that does not activate screen refreshing on dialog termination. Therefore, the displayed file list is not updated when the dialog is closed, and un-hidden files are not displayed. The failure-inducing action is the action that opens the "Hidden Files" dialog (i.e., the one that executes the constructor). However, to determine the failure, after opening the dialog (failure-inducing action), it is necessary to (1) remove from the list of hidden files an hidden file belonging to the current folder, and (2) close the "Hidden Files" dialog before noticing that the un-hidden file is still not shown; in short, when testing is automated by CALM, we have to inspect the screenshot taken after the second action performed after the failure-inducing action.

Table 5.4 shows that nine faults among the eleven selected faults (81.82%) could be detected by Pattern-1 (i.e., inspecting the screenshots recorded before and after the failure-inducing action.) One fault can be detected following Pattern-2 (i.e., inspecting one screenshot taken after an additional action that follows the failure-inducing action). One fault can be detected following Pattern-3 (i.e., inspecting one screenshot taken before an action that follows the failure-inducing action). In conclusion, given the prevalence of Pattern-1, which corresponds to what is detectable with our MTA strategies, our investigation shows that the proposed strategies have the potential to enable the identification of a large proportion of App faults.

5.4.3 RQ2. How effective is CALM in exposing functional faults?

Experiment design

To determine if CALM can select inputs that exercise a fault and make the software fail, we tested the selected subject Apps with CALM and visually inspected the recorded screenshots to determine if a failure was observed. Precisely, for Apps affected by faults that can be detected following Pattern-1, we inspected the screenshots recorded before and after every target action (CALM automatically traces target actions). For Apps affected by faults that can be detected following Pattern-2 or Pattern-3, we also looked for the presence of the additional required screenshot; please note that, as a consequence of the investigation made for RQ1, we know what input can generate the required screenshot, which facilitated our visual inspection (we filtered only the actions matching the required input).

Since previous investigations (see Section 4) have shown that CALM is more effective (code coverage) when updates have a limited magnitude (i.e., a limited number of methods were modified), we assess CALM in two scenarios: (1) when updates have different magnitudes (hereafter, *Config#1*), and (2) when updates have minimal magnitudes (hereafter, *Config#2*). For Config#1, for each selected faulty App version, we applied CALM to test the selected App version, following CALM's workflow; precisely, we incrementally tested with CALM all the versions of the App under test till the faulty version. For Config#2, for each selected faulty App version, we simulated an update of minimal magnitude by collecting the models derived from Config#1 (i.e., the CALM model used to test the faulty App version) and re-executing CALM on the faulty App version after configuring it to treat the faulty method as the only target method. Config#2 enables us to simulate an execution of CALM where the fault has been introduced with a minimal change (i.e., a change of a method that does not affect how to reach the AbstractStates in which the target method is triggered). Config#2 also enables us to simulate a usage scenario where engineers decide to invest additional test budget (i.e., time and hardware resources) to test specific methods.

For each execution of both configurations, as per previous CALM experiments, we allocated a test budget of one hour. To deal with randomness, we tested each faulty version 10 times for each configuration. Then, for each test execution, we determined if CALM led to an App failure, as indicated above. We use the term *Fault Detection Rate (FDR)* to refer the proportion of test executions of faulty Apps in which at least a failure triggered by the fault affecting the App has been observed. Please note that we use the term *fault* because we are verifying if CALM detect at least one failure for the fault affecting the App under test, in other words we aim to verify if CALM enables detecting the fault affecting the App. Furthermore, to better investigate the reasons for failures not being observed, we report the proportion of executions in which the faulty methods have been exercised (hereafter, *Faulty Method Coverage* — FMC).

Results Table 5.5 shows, for each selected fault, the FDR and FMC for each of the two configurations considered.

Please recall that Config#1 includes updates of different magnitude (from 18 to 651 target methods, see Table 5.3), including changes in the Window transitions, while Config#2

Table 5.5: CALM's effectiveness for fault coverage, we report faulty method coverage (FMC) and fault detection rate (FDR), higher values are bold.

BugId	Config#1		Config#2	
	FMC	FDR	FMC	FDR
AC01	10/10 (100%)	0/10 (0%)	10/10 (100%)	**4/10 (40%)**
AC02	9/10 (90%)	7/10 (70%)	9/10 (90%)	**8/10 (80%)**
AC03	6/10 (60%)	6/10 (60%)	6/10 (60%)	6/10 (60%)
AC04	**10/10 (100%)**	5/10 (50%)	9/10 (90%)	**7/10 (70%)**
AM01	10/10 (100%)	4/10 (40%)	10/10 (100%)	**10/10 (100%)**
AM02	10/10 (100%)	4/10 (40%)	10/10 (100%)	**6/10 (60%)**
AM03	10/10 (100%)	9/10 (90%)	10/10 (100%)	**10/10 (100%)**
AM04	0/10(0%)	0/10 (0%)	**10/10 (100%)**	**8/10 (80%)**
AM05	10/10 (100%)	9/10 (90%)	10/10 (100%)	**10/10 (100%)**
AM06	4/10 (40%)	0/10 (0%)	4/10 (40%)	0/10 (0%)
AM07	10/10 (100%)	1/10 (10%)	10/10 (100%)	1/10 (10%)
Average	**81.8%**	**40.9%**	**89.1%**	**63.6%**

does not present any change in the navigation across Windows and the reused model should accurately capture how to test the App. Therefore, with Config#1, it might be more difficult for CALM to reach the Windows that exercise a faulty method than with Config#2; consequently, one may expect that Config#1 should lead to lower FMC than Config#2. However, such situation happens only in the case of AM04, where Config#1 does not enable reaching the faulty method but Config#2 does.

Surprisingly, for one fault (AC04), Config#1 leads to a higher FMC. This happens because the faulty target method is executed only if the App is in a specific App state; such App states are more likely to be selected for Config#1 where a larger number of changes may help CALM exercise a wider set of App states. The FMC of the two configurations match for all the other faults, which means that updates of different magnitude do not prevent CALM from reaching the faulty methods.

Finally, we observe that there is only one faulty method (AM06) that is unlikely reached (i.e., $\text{FMC} < 50\%$) in both configurations. The reason for such low FMC is that such method is reached only with inputs that belong to a narrow proportion of the input space. Specifically, it requires some hidden files (i.e., their file name starts with a dot) on the device, and requires CALM to trigger the function to search for a file using a search string containing part a hidden file's name.

On average, the FDR is 41.91% for Config#1 and 63.64% for Config#2. Expectedly, Config#2 leads to a higher FDR because CALM can invest the whole test budget on the buggy function. For the same reason, two faults (i.e., AC01, AM04) are revealed only with Config#2. Our results suggest that the fault detection capability of CALM can be largely improved by testing, by default, every target function for one hour. If the proposed MTA strategies minimize the number of screenshots to be inspected without compromising failure revealing capabilities, such change in the CALM procedures might be cost-effective even for updates of large magnitude.

5.4.4 RQ3. What is the most cost-effective MTA strategy?

Experiment setup For this research question, we focus on faults for which CALM can trigger a failure (see RQ2) that can be detected with Pattern-1 (see RQ1) because they are the ones targeted by our MTA strategies. It leads to a total of 8 faults.

We evaluate all the MTA strategies described in Table 5.1 in terms of FDR and cost reduction. As per RQ2, FDR captures the proportion of executions in which an MTA strategy selects an action with screenshots enabling fault detection. We compute the FDR obtained for each fault—because fault detection is likely affected by the characteristics of a fault–and we also report the average obtained from the FDR of the 8 selected faults.

Since without a dedicated strategy for the inspection of target actions an engineer should inspect all of them, we compute *cost reduction (CR)* as the proportion of target actions that are not inspected thanks the selected strategy. It is computed as follows

$$CR = 1 - \frac{|\Sigma_{MTA}|}{|\Sigma_F|}$$

For CR, we discuss the results observed for Config#1 and Config#2 separately because the two configurations may lead to faulty methods being exercised with different frequencies (Config#2 focuses only on the faulty method), and, consequently, they may lead to a different number of target actions.

The best approach is the one that achieves a high FDR with a high CR. Table 5.1 shows

Table 5.6: Fault detection rate (%) achieved by each MTA strategy

BugId	AC01	AC02	AC03	AM01	AM02	AM03	AM04	AM05	**Average**		
UTA strategy									**Cfg1**	**Cfg2**	**Overall**
CC_PTIC	0.0	100	100	0.0	100	100	77.8	73.7	**83.3**	**66.0**	**68.9**
CC_WTI_PTIC	0.0	92.9	100	0.0	90.0	100	77.8	73.7	**76.4**	**66.0**	**66.8**
CC_WI_PTIC	0.0	92.9	100	0.0	90.0	100	77.8	68.4	**76.4**	**64.7**	**66.1**
CC_PTIF	0.0	100	100	7.1	100	100	77.8	73.7	**87.5**	**66.0**	**69.8**
CC_WI_PTIF	0.0	100	100	7.1	100	100	77.8	73.7	**87.5**	**66.0**	**69.8**
CC_PIF	50.0	100	100	100	100	100	77.8	100	**100**	**91.0**	**91.0**
AF_SIFT_2M	100	100	100	78.6	40.0	100	100	100	**91.7**	**87.9**	**89.8**
AF_SIFT_3M	100	100	100	85.7	50.0	100	100	100	**91.7**	**91.3**	**92.0**
AF_SSIM_2M	0.0	100	100	85.7	60.0	100	100	100	**91.7**	**80.8**	**80.7**
AF_SSIM_3M	0.0	100	100	85.7	60.0	100	100	100	**91.7**	**80.8**	**80.7**
BC_SIFT_2M_WTI_PTIC	100	100	100	78.6	100	100	100	100	**100**	**96.3**	**97.3**
BC_SIFT_3M_WTI_PTIC	100	100	100	85.7	100	100	100	100	**100**	**97.5**	**98.2**
BC_SSIM_2M_WTI_PTIC	0.0	100	100	85.7	100	100	100	100	**100**	**85.0**	**85.7**
BC_SSIM_3M_WTI_PTIC	0.0	100	100	85.7	100	100	100	100	**100**	**85.0**	**85.7**
BC_SIFT_2M_WI_PTIC	100	100	100	78.6	100	100	100	100	**100**	**96.3**	**97.3**
BC_SIFT_3M_WI_PTIC	100	100	100	85.7	100	100	100	100	**100**	**97.5**	**98.2**
BC_SSIM_2M_WI_PTIC	0.0	100	100	85.7	100	100	100	100	**100**	**85.0**	**85.7**
BC_SSIM_3M_WI_PTIC	0.0	100	100	85.7	100	100	100	100	**100**	**85.0**	**85.7**
BC_SIFT_2M_WI_PTIF	100	100	100	78.6	100	100	100	100	**100**	**96.3**	**97.3**
BC_SIFT_3M_WI_PTIF	100	100	100	85.7	100	100	100	100	**100**	**97.5**	**98.2**
BC_SSIM_2M_WI_PTIF	0.0	100	100	85.7	100	100	100	100	**100**	**85.0**	**85.7**
BC_SSIM_2M_WI_PTIF	0.0	100	100	85.7	100	100	100	100	**100**	**85.0**	**85.7**
BC_SIFT_2M_PIF	100	100	100	100	100	100	100	100	**100**	**100**	**100**
BC_SIFT_3M_PIF	100	100	100	100	100	100	100	100	**100**	**100**	**100**
BC_SSIM_2M_PIF	25.0	100	100	100	100	100	100	100	**100**	**90.6**	**90.6**
BC_SSIM_3M_PIF	25.0	100	100	100	100	100	100	100	**100**	**90.6**	**90.6**

Legend. Cfg1 : Config#1, Cfg2 : Config#2

that the selected strategies cover a large proportion of the possible combination of parameters used to define the algorithms presented in Section 5.3.

We rely on a non-parametric Mann-Whitney U-test [108] and the Vargha and Delaney's A_{12} statistics [100], a non-parametric effect size measure, to determine what MTA strategy performs better. Following standard practice, an MTA strategy is better than another if the difference is statistically significant and $A_{12} > 0.56$. Regarding FDR, an observation is the FDR computed for one fault. For CR, an observation is the CR of one execution of CALM on an App version.

Results Table 5.6 shows FDR results. Strategies based on code coverage (*CC strategies*) yield the worst results, except for CC_PIF. Excluding CC_PIF, they achieve an average FDR between 66.1% to 69.8%; however, if we focus on Config#1 (i.e., the one supported by CALM) they reach up to 87.5% FDR, a high value, thus suggesting that they may still be an acceptable option. Action-effect-based strategies (*AF strategies*) lead to an FDR between 80.7% and 92.0%, which shows that, by focusing on App outputs, we can identify a larger proportion of faults than by relying on most CC strategies. Their FDR goes up to 91% for Config#1. However, combined strategies (CB strategies) yield the best results with an average FDR between 90.6% and 100%, thus implying that CC and AF strategies are complementary.

CC_PIF leads to an average FDR (91%) that is within the range obtained by CB strategies; also, its FDR is equal to 100% for Config#1. Such results are likely due to CC_PIF selecting any action leading to a different set of covered instructions, which may lead to a large set of actions being selected. Consequently, CC_PIF has a higher probability of selecting actions leading to failures but with a high cost, as discussed below.

By analyzing the FDR of each individual fault we may notice that the characteristics of faults seem to affect the performance of each strategy. First, some faults can be detected by most MTA strategies (i.e., Fault AC02, AC03, AM03 with often 100% FDR); these are faults leading to a failure every time the faulty method is executed. One fault (i.e., AM02) is well detected by CC strategies (i.e., 90-100%) but hardly detected with AF strategies (40-60%), which implies that AF strategies do not always select failing actions. It seems to be caused by the state abstraction function of AF strategies that, when applied to menu items, group screenshots exercising different features together, thus rendering AF strategies ineffective when failures affect menu items. Other faults, instead, are better revealed by AF strategies (i.e., AC01, AM01, AM04, AM05) than by CC strategies. These are faults caused by missing code implementing some requirements (e.g., missing conditional statements to handle special cases); CC strategies are not able to discover such cases because, by definition, it is not possible to determine that some desired code is missing by measuring code coverage. This is the case for Fault AC01, which is detected in all our executions (i.e., 4/4) by AF_SIFT_2M and AF_SIFT_3M but not detected by any CC strategy. Figure 5.2 shows a failure caused by Fault AC01, which is caused by missing code (i.e., re-set the color of a Widget to the color selected before screen rotation).

Tables 5.7 and 5.8 provide results for effect size and statistical significance. AF and CB strategies always show a medium or large effect size when compared to CC strategies, except in two cases for AF_SSIM_M2 and AF_SSIM_M3 (small effect size); it indicates that AF

Table 5.7: MannWhitney U-test's P-value for Fault detection rate between each pair of strategy

Strategy		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)	(26)
CC_PTIC	(1)	-	.663	.625	.910	.910	.251	.227	.227	.561	.736	.074	.074	.276	.276	.074	.074	.276	.276	.074	.074	.276	.276	<0.05	<0.05	.125	.125
CC_WTI_PTIC	(2)	.663	-	.957	.588	.588	.082	.072	.072	.271	.387	<0.05	<0.05	.091	.091	<0.05	<0.05	.091	.091	<0.05	<0.05	.091	.091	<0.05	<0.05	<0.05	<0.05
CC_WI_PTIC	(3)	.625	.957	-	.552	.552	.082	.072	.072	.271	.387	<0.05	<0.05	.091	.091	<0.05	<0.05	.091	.091	<0.05	<0.05	.091	.091	<0.05	<0.05	<0.05	<0.05
CC_PTIF	(4)	.910	.588	.552	-	1.0	.251	.228	.228	.602	.779	.074	.074	.305	.305	.074	.074	.305	.305	.074	.074	.305	.305	<0.05	<0.05	.125	.125
CC_WI_PTIF	(5)	.910	.588	.552	1.0	-	.251	.228	.228	.602	.779	.074	.074	.305	.305	.074	.074	.305	.305	.074	.074	.305	.305	<0.05	<0.05	.125	.125
CC_PIF	(6)	.251	.082	.082	.251	.251	-	1.0	.945	.610	.398	.441	.441	1.0	1.0	.441	.441	1.0	1.0	.441	.441	1.0	1.0	.144	.144	.644	.644
AF_SIFT_2M	(7)	.227	.072	.072	.228	.228	1.0	-	.890	.610	.398	.487	.441	1.0	1.0	.487	.441	1.0	1.0	.487	.441	1.0	1.0	.144	.144	.644	.644
AF_SIFT_3M	(8)	.227	.072	.072	.228	.228	.945	.890	-	.565	.365	.538	.487	.945	.945	.538	.487	.945	.945	.538	.487	.945	.945	.144	.144	.644	.644
AF_SSIM_2M	(9)	.561	.271	.271	.602	.602	.610	.610	.565	-	.772	.241	.214	.609	.609	.241	.214	.609	.609	.241	.214	.609	.609	.064	.064	.301	.301
AF_SSIM_3M	(10)	.736	.387	.387	.779	.779	.398	.398	.365	.772	-	.125	.110	.397	.397	.125	.110	.397	.397	.125	.110	.397	.397	<0.05	<0.05	.160	.160
BC_SIFT_2M_WTI_PTIC	(12)	.074	<0.05	<0.05	.074	.074	.441	.487	.538	.241	.125	-	.927	.538	.538	1.0	.927	.538	.538	1.0	.927	.538	.538	.317	.317	.927	.927
BC_SIFT_3M_WTI_PTIC	(13)	.074	<0.05	<0.05	.074	.074	.441	.441	.487	.214	.110	.927	-	.487	.487	.927	1.0	.487	.487	.927	1.0	.487	.487	.317	.317	.927	.927
BC_SSIM_2M_WTI_PTIC	(14)	.276	.091	.091	.305	.305	1.0	1.0	.945	.609	.397	.538	.487	-	1.0	.538	.487	1.0	1.0	.538	.487	1.0	1.0	.144	.144	.538	.538
BC_SSIM_3M_WTI_PTIC	(15)	.276	.091	.091	.305	.305	1.0	1.0	.945	.609	.397	.538	.487	1.0	-	.538	.487	1.0	1.0	.538	.487	1.0	1.0	.144	.144	.538	.538
BC_SIFT_2M_WI_PTIC	(16)	.074	<0.05	<0.05	.074	.074	.441	.487	.538	.241	.125	1.0	.927	.538	.538	-	.927	.538	.538	1.0	.927	.538	.538	.317	.317	.927	.927
BC_SIFT_3M_WI_PTIC	(17)	.074	<0.05	<0.05	.074	.074	.441	.441	.487	.214	.110	.927	1.0	.487	.487	.927	-	.487	.487	.927	1.0	.487	.487	.317	.317	.927	.927
BC_SSIM_2M_WI_PTIC	(18)	.276	.091	.091	.305	.305	1.0	1.0	.945	.609	.397	.538	.487	1.0	1.0	.538	.487	-	1.0	.538	.487	1.0	1.0	.144	.144	.538	.538
BC_SSIM_3M_WI_PTIC	(19)	.276	.091	.091	.305	.305	1.0	1.0	.945	.609	.397	.538	.487	1.0	1.0	.538	.487	1.0	-	.538	.487	1.0	1.0	.144	.144	.538	.538
BC_SIFT_2M_WI_PTIF	(20)	.074	<0.05	<0.05	.074	.074	.441	.487	.538	.241	.125	1.0	.927	.538	.538	1.0	.927	.538	.538	-	.927	.538	.538	.317	.317	.927	.927
BC_SIFT_3M_WI_PTIF	(21)	.074	<0.05	<0.05	.074	.074	.441	.441	.487	.214	.110	.927	1.0	.487	.487	.927	1.0	.487	.487	.927	-	.487	.487	.317	.317	.927	.927
BC_SSIM_2M_WI_PTIF	(22)	.276	.091	.091	.305	.305	1.0	1.0	.945	.609	.397	.538	.487	1.0	1.0	.538	.487	1.0	1.0	.538	.487	-	1.0	.144	.144	.538	.538
BC_SSIM_2M_WI_PTIF	(23)	.276	.091	.091	.305	.305	1.0	1.0	.945	.609	.397	.538	.487	1.0	1.0	.538	.487	1.0	1.0	.538	.487	1.0	-	.144	.144	.538	.538
BC_SIFT_2M_PIF	(24)	<0.05	<0.05	<0.05	<0.05	<0.05	.144	.144	.144	.064	<0.05	.317	.317	.144	.144	.317	.317	.144	.144	.317	.317	.144	.144	-	<0.05	.317	.317
BC_SIFT_3M_PIF	(25)	<0.05	<0.05	<0.05	<0.05	<0.05	.144	.144	.144	.064	<0.05	.317	.317	.144	.144	.317	.317	.144	.144	.317	.317	.144	.144	<0.05	-	.317	.317
BC_SSIM_2M_PIF	(26)	.125	<0.05	<0.05	.125	.125	.644	.644	.644	.301	.160	.927	.927	.538	.538	.927	.927	.538	.538	.927	.927	.538	.538	.317	.317	-	1.0
BC_SSIM_3M_PIF	(27)	.125	<0.05	<0.05	.125	.125	.644	.644	.644	.301	.160	.927	.927	.538	.538	.927	.927	.538	.538	.927	.927	.538	.538	.317	.317	1.0	-

Table 5.8: A_{12} effect size for Fault detection rate between each pair of strategy

Strategy		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)	(26)
CC_PTIC	(1)	-	.563	.570	.484	.484	.352	.344	.344	.422	.453	.281	.281	.359	.359	.281	.281	.359	.359	.281	.281	.359	.359	.250	.250	.313	.313
CC_WTI_PTIC	(2)	.438	-	.508	.422	.422	.258	.250	.250	.344	.375	.172	.172	.266	.266	.172	.172	.266	.266	.172	.172	.266	.266	.125	.125	.203	.203
CC_WI_PTIC	(3)	.430	.492	-	.414	.414	.258	.250	.250	.344	.375	.172	.172	.266	.266	.172	.172	.266	.266	.172	.172	.266	.266	.125	.125	.203	.203
CC_PTIF	(4)	.516	.578	.586	-	.500	.352	.344	.344	.430	.461	.281	.281	.367	.367	.281	.281	.367	.367	.281	.281	.367	.367	.250	.250	.313	.313
CC_WI_PTIF	(5)	.516	.578	.586	.500	-	.352	.344	.344	.430	.461	.281	.281	.367	.367	.281	.281	.367	.367	.281	.281	.367	.367	.250	.250	.313	.313
CC_PIF	(6)	.648	.742	.742	.648	.648	-	.500	.492	.563	.609	.422	.422	.500	.500	.422	.422	.500	.500	.422	.422	.500	.500	.375	.375	.453	.453
AF_SIFT_2M	(7)	.656	.750	.750	.656	.656	.500	-	.484	.563	.609	.430	.422	.500	.500	.430	.422	.500	.500	.430	.422	.500	.500	.375	.375	.453	.453
AF_SIFT_3M	(8)	.656	.750	.750	.656	.656	.508	.516	-	.570	.617	.438	.430	.508	.508	.438	.430	.508	.508	.438	.430	.508	.508	.375	.375	.453	.453
AF_SSIM_2M	(9)	.578	.656	.656	.570	.570	.438	.438	.430	-	.539	.367	.359	.438	.438	.367	.359	.438	.438	.367	.359	.438	.438	.313	.313	.383	.383
AF_SSIM_3M	(10)	.547	.625	.625	.539	.539	.391	.391	.383	.461	-	.313	.305	.391	.391	.313	.305	.391	.391	.313	.305	.391	.391	.250	.250	.328	.328
BC_SIFT_2M_WTI_PTIC	(12)	.719	.828	.828	.719	.719	.578	.570	.563	.633	.688	-	.492	.563	.563	.500	.492	.563	.563	.500	.492	.563	.563	.438	.438	.508	.508
BC_SIFT_3M_WTI_PTIC	(13)	.719	.828	.828	.719	.719	.578	.578	.570	.641	.695	.508	-	.570	.570	.508	.500	.570	.570	.508	.500	.570	.570	.438	.438	.508	.508
BC_SSIM_2M_WTI_PTIC	(14)	.641	.734	.734	.633	.633	.500	.500	.492	.563	.609	.438	.430	-	.500	.438	.430	.500	.500	.438	.430	.500	.500	.375	.375	.438	.438
BC_SSIM_3M_WTI_PTIC	(15)	.641	.734	.734	.633	.633	.500	.500	.492	.563	.609	.438	.430	.500	-	.438	.430	.500	.500	.438	.430	.500	.500	.375	.375	.438	.438
BC_SIFT_2M_WI_PTIC	(16)	.719	.828	.828	.719	.719	.578	.570	.563	.633	.688	.500	.492	.563	.563	-	.492	.563	.563	.500	.492	.563	.563	.438	.438	.508	.508
BC_SIFT_3M_WI_PTIC	(17)	.719	.828	.828	.719	.719	.578	.578	.570	.641	.695	.508	.500	.570	.570	.508	-	.570	.570	.508	.500	.570	.570	.438	.438	.508	.508
BC_SSIM_2M_WI_PTIC	(18)	.641	.734	.734	.633	.633	.500	.500	.492	.563	.609	.438	.430	.500	.500	.438	.430	-	.500	.438	.430	.500	.500	.375	.375	.438	.438
BC_SSIM_3M_WI_PTIC	(19)	.641	.734	.734	.633	.633	.500	.500	.492	.563	.609	.438	.430	.500	.500	.438	.430	.500	-	.438	.430	.500	.500	.375	.375	.438	.438
BC_SIFT_2M_WI_PTIF	(20)	.719	.828	.828	.719	.719	.578	.570	.563	.633	.688	.500	.492	.563	.563	.500	.492	.563	.563	-	.492	.563	.563	.438	.438	.508	.508
BC_SIFT_3M_WI_PTIF	(21)	.719	.828	.828	.719	.719	.578	.578	.570	.641	.695	.508	.500	.570	.570	.508	.500	.570	.570	.508	-	.570	.570	.438	.438	.508	.508
BC_SSIM_2M_WI_PTIF	(22)	.641	.734	.734	.633	.633	.500	.500	.492	.563	.609	.438	.430	.500	.500	.438	.430	.500	.500	.438	.430	-	.500	.375	.375	.438	.438
BC_SSIM_2M_WI_PTIF	(23)	.641	.734	.734	.633	.633	.500	.500	.492	.563	.609	.438	.430	.500	.500	.438	.430	.500	.500	.438	.430	.500	-	.375	.375	.438	.438
BC_SIFT_2M_PIF	(24)	.750	.875	.875	.750	.750	.625	.625	.625	.688	.750	.563	.563	.625	.625	.563	.563	.625	.625	.563	.563	.625	.625	-	.500	.563	.563
BC_SIFT_3M_PIF	(25)	.750	.875	.875	.750	.750	.625	.625	.625	.688	.750	.563	.563	.625	.625	.563	.563	.625	.625	.563	.563	.625	.625	.500	-	.563	.563
BC_SSIM_2M_PIF	(26)	.688	.797	.797	.688	.688	.547	.547	.547	.617	.672	.492	.492	.563	.563	.492	.492	.563	.563	.492	.492	.563	.563	.438	.438	-	.500
BC_SSIM_3M_PIF	(27)	.688	.797	.797	.688	.688	.547	.547	.547	.617	.672	.492	.492	.563	.563	.492	.492	.563	.563	.492	.492	.563	.563	.438	.438	.500	-

Table 5.9: MannWhitney U-test's P-value for Cost reduction between each pair of strategy

Strategy		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)	(26)
CC_PTIC	(1)	-	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
CC_WTI_PTIC	(2)	<0.05	-	.451	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
CC_WI_PTIC	(3)	<0.05	.451	-	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
CC_PTIF	(4)	<0.05	<0.05	<0.05	-	1.0	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
CC_WI_PTIF	(5)	<0.05	<0.05	<0.05	1.0	-	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
CC_PIF	(6)	<0.05	<0.05	<0.05	<0.05	<0.05	-	<0.05	.151	<0.05	.196	<0.05	.371	<0.05	.461	<0.05	.371	<0.05	.461	.377	.134	.424	.095	<0.05	<0.05	<0.05	<0.05
AF_SIFT_2M	(7)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	-	<0.05	.989	<0.05	.146	<0.05	.122	<0.05	.145	<0.05	.119	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
AF_SIFT_3M	(8)	<0.05	<0.05	<0.05	<0.05	<0.05	.151	<0.05	-	<0.05	.755	<0.05	.256	<0.05	.170	<0.05	.256	<0.05	.170	.450	<0.05	.403	<0.05	<0.05	<0.05	<0.05	<0.05
AF_SSIM_2M	(9)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.989	<0.05	-	<0.05	.159	<0.05	.126	<0.05	.158	<0.05	.120	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
AF_SSIM_3M	(10)	<0.05	<0.05	<0.05	<0.05	<0.05	.196	<0.05	.755	<0.05	-	<0.05	.383	<0.05	.252	<0.05	.358	<0.05	.252	.578	<0.05	.495	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SIFT_2M_WTI_PTIC	(12)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.146	<0.05	.159	<0.05	-	<0.05	.997	<0.05	.946	<0.05	.994	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SIFT_3M_WTI_PTIC	(13)	<0.05	<0.05	<0.05	<0.05	<0.05	.371	<0.05	.256	<0.05	.383	<0.05	-	<0.05	.830	<0.05	.934	<0.05	.762	.941	<0.05	.929	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SSIM_2M_WTI_PTIC	(14)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.122	<0.05	.126	<0.05	.997	<0.05	-	<0.05	.992	<0.05	.951	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SSIM_3M_WTI_PTIC	(15)	<0.05	<0.05	<0.05	<0.05	<0.05	.461	<0.05	.170	<0.05	.252	<0.05	.830	<0.05	-	<0.05	.851	<0.05	.951	.881	<0.05	.874	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SIFT_2M_WI_PTIC	(16)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.145	<0.05	.158	<0.05	.946	<0.05	.992	<0.05	-	<0.05	.998	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SIFT_3M_WI_PTIC	(17)	<0.05	<0.05	<0.05	<0.05	<0.05	.371	<0.05	.256	<0.05	.358	<0.05	.934	<0.05	.851	<0.05	-	<0.05	.830	.951	<0.05	.929	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SSIM_2M_WI_PTIC	(18)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.119	<0.05	.120	<0.05	.994	<0.05	.951	<0.05	.998	<0.05	-	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SSIM_3M_WI_PTIC	(19)	<0.05	<0.05	<0.05	<0.05	<0.05	.461	<0.05	.170	<0.05	.252	<0.05	.762	<0.05	.951	<0.05	.830	<0.05	-	.881	<0.05	.874	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SIFT_2M_WI_PTIF	(20)	<0.05	<0.05	<0.05	<0.05	<0.05	.377	<0.05	.450	<0.05	.578	<0.05	.941	<0.05	.881	<0.05	.951	<0.05	.881	-	<0.05	.848	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SIFT_3M_WI_PTIF	(21)	<0.05	<0.05	<0.05	<0.05	<0.05	.134	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	-	<0.05	.843	<0.05	<0.05	<0.05	<0.05
BC_SSIM_2M_WI_PTIF	(22)	<0.05	<0.05	<0.05	<0.05	<0.05	.424	<0.05	.403	<0.05	.495	<0.05	.929	<0.05	.874	<0.05	.929	<0.05	.874	.848	<0.05	-	<0.05	<0.05	<0.05	<0.05	<0.05
BC_SSIM_2M_WI_PTIF	(23)	<0.05	<0.05	<0.05	<0.05	<0.05	.095	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.843	<0.05	-	<0.05	<0.05	<0.05	<0.05
BC_SIFT_2M_PIF	(24)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	-	.133	.910	.093
BC_SIFT_3M_PIF	(25)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.133	-	.154	.795
BC_SSIM_2M_PIF	(26)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.910	.154	-	.109
BC_SSIM_3M_PIF	(27)	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	<0.05	.093	.795	.109	-

Table 5.10: A_{12} effect size for Cost reduction between each pair of strategy

Strategy		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)	(26)
CC_PTIC	(1)	-	.419	.436	.641	.641	.834	.870	.895	.868	.895	.873	.896	.870	.895	.873	.896	.870	.895	.878	.899	.876	.899	.904	.911	.903	.911
CC_WTI_PTIC	(2)	.581	-	.522	.660	.660	.857	.881	.902	.880	.903	.884	.903	.883	.904	.884	.903	.883	.904	.890	.907	.889	.908	.911	.918	.911	.918
CC_WI_PTIC	(3)	.564	.478	-	.652	.652	.855	.879	.901	.877	.901	.882	.901	.880	.902	.882	.901	.880	.902	.888	.905	.886	.906	.909	.917	.909	.917
CC_PTIF	(4)	.359	.340	.348	-	.500	.752	.668	.750	.670	.752	.696	.775	.697	.780	.696	.775	.697	.780	.755	.801	.757	.803	.816	.836	.816	.838
CC_WI_PTIF	(5)	.359	.340	.348	.500	-	.752	.668	.750	.670	.752	.696	.775	.697	.780	.696	.775	.697	.780	.755	.801	.757	.803	.816	.836	.816	.838
CC_PIF	(6)	.166	.143	.145	.248	.248	-	.389	.459	.391	.463	.402	.474	.405	.479	.402	.474	.405	.479	.475	.543	.477	.548	.616	.642	.617	.644
AF_SIFT_2M	(7)	.130	.119	.121	.332	.332	.611	-	.619	.500	.624	.542	.637	.544	.643	.542	.637	.545	.643	.616	.677	.616	.680	.709	.739	.707	.742
AF_SIFT_3M	(8)	.105	.098	.099	.250	.250	.541	.381	-	.382	.509	.420	.532	.422	.539	.420	.532	.422	.539	.522	.599	.524	.607	.651	.683	.651	.685
AF_SSIM_2M	(9)	.132	.120	.123	.330	.330	.609	.500	.618	-	.623	.540	.636	.544	.642	.540	.636	.544	.642	.611	.674	.614	.679	.707	.736	.706	.739
AF_SSIM_3M	(10)	.105	.097	.099	.248	.248	.537	.376	.491	.377	-	.413	.525	.415	.533	.413	.526	.415	.533	.516	.594	.520	.601	.646	.680	.647	.682
BC_SIFT_2M_WTI_PTIC	(12)	.127	.116	.118	.304	.304	.598	.458	.580	.460	.587	-	.606	.500	.611	.502	.606	.500	.612	.591	.658	.591	.665	.694	.724	.693	.728
BC_SIFT_3M_WTI_PTIC	(13)	.104	.097	.099	.225	.225	.526	.363	.468	.364	.475	.394	-	.392	.506	.394	.502	.394	.509	.502	.580	.503	.586	.639	.672	.638	.675
BC_SSIM_2M_WTI_PTIC	(14)	.130	.117	.120	.303	.303	.595	.456	.578	.456	.585	.500	.608	-	.611	.500	.608	.502	.612	.587	.657	.590	.663	.694	.724	.693	.727
BC_SSIM_3M_WTI_PTIC	(15)	.105	.096	.098	.220	.220	.521	.357	.461	.358	.467	.389	.494	.389	-	.389	.495	.389	.502	.496	.573	.495	.578	.634	.668	.634	.672
BC_SIFT_2M_WI_PTIC	(16)	.127	.116	.118	.304	.304	.598	.458	.580	.460	.587	.498	.606	.500	.611	-	.606	.500	.612	.591	.657	.591	.664	.694	.724	.693	.728
BC_SIFT_3M_WI_PTIC	(17)	.104	.097	.099	.225	.225	.526	.363	.468	.364	.474	.394	.498	.392	.505	.394	-	.393	.506	.502	.580	.503	.585	.639	.672	.638	.675
BC_SSIM_2M_WI_PTIC	(18)	.130	.117	.120	.303	.303	.595	.455	.578	.456	.585	.500	.606	.498	.611	.500	.607	-	.612	.587	.656	.590	.663	.693	.724	.692	.727
BC_SSIM_3M_WI_PTIC	(19)	.105	.096	.098	.220	.220	.521	.357	.461	.358	.467	.388	.491	.388	.498	.388	.494	.388	-	.496	.573	.495	.578	.634	.668	.634	.672
BC_SIFT_2M_WI_PTIF	(20)	.122	.110	.112	.245	.245	.525	.384	.478	.389	.484	.409	.498	.413	.504	.409	.498	.413	.504	-	.571	.505	.579	.641	.677	.641	.681
BC_SIFT_3M_WI_PTIF	(21)	.101	.093	.095	.199	.199	.457	.323	.401	.326	.406	.342	.420	.343	.427	.343	.420	.344	.427	.429	-	.432	.506	.588	.623	.590	.626
BC_SSIM_2M_WI_PTIF	(22)	.124	.111	.114	.243	.243	.523	.384	.476	.386	.480	.409	.497	.410	.505	.409	.497	.410	.505	.495	.568	-	.577	.638	.676	.639	.679
BC_SSIM_2M_WI_PTIF	(23)	.101	.092	.094	.197	.197	.452	.320	.393	.321	.399	.335	.414	.337	.422	.336	.415	.337	.422	.421	.494	.423	-	.583	.620	.586	.623
BC_SIFT_2M_PIF	(24)	.096	.089	.091	.184	.184	.384	.291	.349	.293	.354	.306	.361	.306	.366	.306	.361	.307	.366	.359	.412	.362	.417	-	.543	.503	.548
BC_SIFT_3M_PIF	(25)	.089	.082	.083	.164	.164	.358	.261	.317	.264	.320	.276	.328	.276	.332	.276	.328	.276	.332	.323	.377	.324	.380	.457	-	.459	.507
BC_SSIM_2M_PIF	(26)	.097	.089	.091	.184	.184	.383	.293	.349	.294	.353	.307	.362	.307	.366	.307	.362	.308	.366	.359	.410	.361	.414	.497	.541	-	.546
BC_SSIM_3M_PIF	(27)	.089	.082	.083	.162	.162	.356	.258	.315	.261	.318	.272	.325	.273	.328	.272	.325	.273	.328	.319	.374	.321	.377	.452	.493	.454	-

and CB strategies are more likely to lead to better results than CC strategies. Unfortunately, most of the p-values derived using the U-test are high (i.e., >0.05), which indicates that we cannot claim any difference between the distribution obtained by these approaches; however, such high p-values are likely due to the limited number of datapoints considered (8 in total, as we compute one FDR value for each fault). Finally, the best performing strategies (i.e., BC_SIFT_2M_PIF and BC_SIFT_3M_PIF) show the lowest p-values and highest effect size, even when compared with other CB strategies, which highlights that they have a slightly better performance.

Table 5.11 shows cost reduction results. First, we discuss overall results across both Config#1 and Config#2. CC strategies focusing on target instructions are very effective, with 84% to 85.6% cost reduction, on average. Instead, CC strategies relying on instruction footprints (i.e., CC_PTIF, CC_WI_PTIF, and CC_PIF) yield a lower cost reduction. AF strategies lead to a cost reduction between 53.6% and 60.4%, thus performing better than CC_PIF but worse than other CC strategies. For combined strategies, the cost reduction decreases, but the difference in performance varies based on the selected CC strategy. For CB strategies combining the p_{tic} property with greedy strategies based on either ω_{ti} of ω_i, cost reduction ranges between 56.4 and 62.4. For CB strategies relying on target instruction footprint properties (i.e., p_{tif}), cost reduction ranges between 56.4 and 62.4. For CB strategies relying on instruction footprint properties (i.e., p_{if}), cost reduction is the worst, between 38.7 and 42.1.

Table 5.9 and Table 5.10 provide results for statistical significance and effect size; thanks to the large number of collected datapoints (we have 101 datapoints, one for each fault being tested in a distinct run), differences are always significant. In addition to that, our results further highlight that CC strategies tend to have a higher cost reduction rate than other strategies (large effect size).

The differences in cost between Config#1 and Config#2 are limited in terms of ranking (i.e., rankings are the same for most of the approaches); however, CC_PIF and AF_SSIM_3M perform better with Config#1, while CC strategies relying on p_{tic} perform better with Config#2. If we look at the cost of each approach (i.e., number of actions to inspect), we can observe that the differences between the different strategies are much more tangible for Config#1 than for Config#2. Indeed for Config#2 the best approach required the inspection

Table 5.11: Cost and cost reduction achieved by each MTA strategy for Config-1, Config-2. Average results are computed across data points. "CR" stands for Cost Reduction. "R" provides the ranking of each strategy.

UTA strategy	Config#1			Config#2			**Average**		
	Cost	CR	R	Cost	CR	R	Cost	CR	R
CC_PTIC	107.4	88.0	3	2.4	79.4	3	**59.0**	**84.0**	3
CC_WTI_PTIC	86.8	90.3	1	2.3	80.1	1	**47.8**	**85.6**	1
CC_WI_PTIC	88.3	90.1	2	2.7	79.6	2	**48.9**	**85.3**	2
CC_PTIF	267.1	69.8	4	2.8	78.9	4	**145.3**	**74.0**	4
CC_WI_PTIF	267.1	69.8	4	2.8	78.9	4	**145.3**	**74.0**	4
CC_PIF	395.1	57.5	14	21.2	45.5	22	**222.8**	**52.0**	20
AF_SIFT_2M	357.3	65.0	7	12.9	55.0	6	**198.6**	**60.4**	6
AF_SIFT_3M	421.7	58.8	12	16.3	48.3	14	**234.9**	**53.9**	12
AF_SSIM_2M	357.6	65.0	6	12.9	54.8	7	**198.7**	**60.3**	7
AF_SSIM_3M	422.9	58.7	13	16.3	47.7	18	**235.6**	**53.6**	13
BC_SIFT_2M_WTI_PTIC	381.0	62.4	8	13.1	54.5	8	**211.5**	**58.8**	8
BC_SIFT_3M_WTI_PTIC	441.7	56.6	15	16.5	47.9	15	**245.8**	**52.6**	16
BC_SSIM_2M_WTI_PTIC	381.4	62.4	10	13.1	54.3	11	**211.7**	**58.7**	10
BC_SSIM_3M_WTI_PTIC	443.2	56.5	17	16.5	47.4	19	**246.6**	**52.3**	18
BC_SIFT_2M_WI_PTIC	381.3	62.4	9	13.1	54.5	8	**211.6**	**58.8**	9
BC_SIFT_3M_WI_PTIC	442.0	56.5	16	16.5	47.9	15	**245.9**	**52.6**	17
BC_SSIM_2M_WI_PTIC	381.7	62.4	11	13.1	54.3	11	**211.8**	**58.7**	11
BC_SSIM_3M_WI_PTIC	443.4	56.4	18	16.5	47.4	19	**246.7**	**52.3**	19
BC_SIFT_2M_WI_PTIF	465.4	52.8	19	13.1	54.5	10	**257.0**	**53.6**	14
BC_SIFT_3M_WI_PTIF	512.7	48.5	21	16.5	47.9	15	**284.1**	**48.3**	21
BC_SSIM_2M_WI_PTIF	466.3	52.8	20	13.1	54.3	13	**257.5**	**53.5**	15
BC_SSIM_2M_WI_PTIF	514.2	48.4	22	16.5	47.3	21	**284.9**	**47.9**	22
BC_SIFT_2M_PIF	535.2	45.9	23	24.0	37.7	23	**299.6**	**42.1**	23
BC_SIFT_3M_PIF	570.5	42.8	25	25.5	34.7	25	**319.4**	**39.0**	25
BC_SSIM_2M_PIF	536.5	45.8	24	24.0	37.5	24	**300.3**	**42.0**	24
BC_SSIM_3M_PIF	572.4	42.6	26	25.6	34.1	26	**320.4**	**38.7**	26

of an average of 2.3 actions while the worst performing approach required the inspection of 25.6 actions; assuming one minute being required for the inspection of each action, it is less than half an hour, which is tangible but not detrimental to the testing process. For Config#1, instead, the difference between the best and the worst approach is 485.6 actions (i.e., $572.4 - 86.8$), leading to eight hours of difference, an practically significant difference (e.g., leading to delay the software release by one working day). Based on such costs, the benefits provided by CC strategies focusing on target instructions are clearly tangible, even if they do not lead to detecting all the faults.

If we consider both cost and effectiveness, our findings indicate that, except for greedy strategies maximizing instruction footprint, *CC strategies lead to sub-optimal fault detection effectiveness* (i.e., FDR between 65% and 70% across the two configurations, but between 76.4% and 87.5% for Config#1); however, they have a limited cost, with 80 to 110 actions to be inspected for Config#1, in our experiments. Overall, the strategy minimizing costs is CC_WTI_PTIC, but has the lowest FDR for coverage strategies. The highest FDR is obtained by CC_PTIF and CC_WI_PTIF but they have the highest cost, especially for Config#1 (above 250 actions to be inspected). The *best cost-effectiveness compromise for CC strategies is obtained with CC_PTIC*, which achieves the second highest FDR (83.3% for Config1) with the third lowest cost (less than 110 actions for Config1, which may lead to three hours and half of inspection, an acceptable effort). Since CC_PTIC matches the UTA selection strategy implemented originally in CALM, our results show that the findings of previous empirical assessments based on CC_PTIC remain valid, from a practical standpoint. *Action-effect-based strategies are more effective than coverage-based strategies for fault detection*; however, they reduce costs only by 50% to 60%. AF_SIFT_2M is the strategy leading to a reasonably high fault detection rate (>85%) with an average of 357 actions to be inspected for Config#1; given that AF_SIFT_2M requires an average of 70 actions fewer than the most effective strategy (i.e., AF_SIFT_3M), we can conclude that it is a good compromise for AF strategies. The highest FDRs are obtained by combined strategies; for example, *the highest FDR, with the higher CR, is obtained by BC_SIFT_2M_PIF*, which detects all the faults, but has one of the highest costs, with an average of 535 actions to be inspected for Config#1. Concluding, if cost minimization must be prioritized, then CC_PTIC is the best strategy; otherwise, if cost is not a major issue (e.g., assessment is provided by crowdworkers [80]), then BC_SIFT_2M_PIF should be preferred.

5.4.5 Threats to Validity

Internal validity. We manually tested each subject to ensure that failures are reproducible. Also, we used a stable version of CALM assessed in the empirical evaluation of Chapter 4. Further, we carefully inspected CALM results to determine the presence of faults preventing the observation of failures in the subject Apps. Finally, we carefully tested all our MTA strategy implementations.

Conclusion validity. To avoid violating the assumptions of parametric statistical tests, we rely on a non-parametric test and effect size measure (i.e., Mann Whitney U-test and the Vargha and Delaney's A_{12} statistics, respectively).

Construct validity. The main constructs considered in this work are fault detection rate and cost reduction. Fault detection is one of the ultimate goals of testing and can be directly measured in the presence of Apps with known faults. Cost, in our context, depends on the visual inspection of App outputs; therefore, assuming a similar cost for inspecting the output of each action (see discussion about *Human effort* in Section 4.3), we can rely on the number of actions to be inspected to compare the cost of different MTA strategies.

External validity. We considered 11 real functional faults from two open-source Apps that are popular among Android users and frequently used as subject Apps in empirical assessments of automated testing techniques [13, 22, 30, 124]. The considered faults lead to failures that can be grouped into two categories (i.e., *UI display issue* and *UI interaction issue*) that, according to a recent study [31], are observed in 97% of Android functional faults. Therefore, although the number of faults selected for our study is limited, they should be representative of a large variety of Android faults.

5.5 Conclusion

To minimize oracle cost in automated App testing, we proposed different strategies for generating minimized sets of target actions (i.e., test actions that exercise the methods modified by an updated App version); we refer to them as MTA strategies. The actions selected by MTA strategies can be presented to software engineers along with the screenshots of the App screen before and after executing a test action; visual inspection should enable engineers to determine if the App failed or not.

Our MTA strategies either rely on code coverage (coverage-based strategies), action effect (action-effect-based strategies), or both (combined strategies). Coverage-based strategies can rely on three distinct coverage properties (i.e., increasing coverage of updated methods, covering distinct sets of instructions in the whole App or in updated methods only) to select actions. Also, to reduce the number of selected actions, we either rely on a chronological approach (i.e., following the order in test input sequences) or a greedy approach (i.e., by maximizing the coverage of instructions belonging to methods modified by the update, or the overall number of instructions). The action-effect-based strategies rely on screenshot distances to measure action effects (i.e., SSIM, SIFT). Combined strategies first employ an action-effect-based strategy, then a coverage-based strategy to complement the action set acquired by the former.

Our empirical evaluation shows that, by visualizing the screenshots taken before and after target actions, our MTA strategies may enable the identification of a large proportion of App faults (81.82%, in our subjects). Also, CALM can successfully exercise failures in 41% of the testing sessions and its results can be largely improved (up to 63.6%) by testing each target function for one hour. Finally, in terms of MTA strategies, our results show that our strategy relying on selecting the actions that contribute to coverage of updated methods, provides the highest cost reduction, with a sufficiently high fault detection rate (68.9%). Instead, to maximize fault detection, engineers should rely on a combined strategy of selecting actions leading to the maximal and the minimal difference between screenshots before and after the action, based on SIFT distance and covering distinct sets of instructions in the whole App (100% of fault detection rate).

Chapter 6

Conclusion & Future Prospects

6.1 Conclusion

In this PhD dissertation, we addressed the problem of automatically testing App updates in a cost-effective way. First, we presented *ATUA (Automated Testing of Updates for App)*, which relies on both static and dynamic analyses, and integrates different testing strategies to generate test inputs focusing on updated methods. Furthermore, we introduced *CALM (Continous Adaptation of Learned Models)*, which enables the efficient reuse of models derived from the testing of previous App versions to identify paths to GUI components that are more likely to exercise updated methods effectively. Finally, we address the oracle problem by defining strategies for selecting a minimal set of test outputs (i.e., screenshots) that potentially include faults, to be provided to engineers for visual inspection, thus saving significant human effort.

Our empirical evaluations are conducted on popular open-source and commercial Apps, with varied magnitudes of updates between versions. The results show that the proposed automated testing approaches (i.e., ATUA, CALM) lead to higher coverage of updated methods and instructions than SOTA approaches for the same human effort in terms of oracles. By reusing the model obtained from testing previous App versions for minor updates, which are common in the CI context, CALM results in a quicker coverage increase in terms of updated instructions. A preliminary assessment of CALM on a set of real faults from open-source Apps suggests that CALM's failure detection capability could be improved by allocating relevant test budgets for testing each updated method, which is scalable in practice. Moreover, the evaluation shows that, by visualizing the screenshots taken before and after target actions, our filtering strategies lead to the identification of a large proportion of App faults. Last, our results show that the proposed strategies could achieve a fault detection rate of 100% with significant cost reduction (i.e., 42%).

In our experiments, we considered only Android Apps, which is standard practice in most App testing research papers. The prevalence of Android in research papers is primarily due to its worldwide dissemination and the availability of a more extensive set of tools to test and analyze Android executable bytecode [2]. In our work, the choice of relying on Android Apps enabled the comparison of ATUA and CALM with state-of-the-art tools working for Android Apps (i.e., Monkey, APE, DM2, Fastbot2, Humandnoid, and Timemachine). However, our approach does not rely on any assumption restricting its applicability to Android. To drive testing, it requires code coverage, which is measurable on any platform, and GUI Trees. CALM extracts GUI Trees by relying on the Android UIAutomator API. Similar features are provided by Appium [125], which works with both iOS and Windows OS. Also, Harmony OS may provide a UIAutomator-like API.

6.2 Future Prospects

In Chapter 5, we focus on Pattern-1 faults, which if present, can be identified by inspecting the screenshots taken before and after the action exercising the faulty code since they are the ones suggested in the rest of the thesis and are the most common inspection pattern according to our study. Our future work may expand the set of subjects to obtain additional cases for other inspection patterns requiring additional screenshots recorded when an action before the failure-inducing action is triggered (Pattern-2) or after the failure-inducing action is triggered (Pattern-3). Further, we desire to strengthen the results in Chapter 5 through a more extensive empirical evaluation that potentially includes closed-source Apps if their developers provide access to bug information. Closed-source Apps usually consist of more complex UI and functions, so they are potentially relevant for assessing the MTA strategies. Open-source candidates for the evaluation could be retrieved from existing benchmarks [126, 31]. Further, instead of assessing all the outputs, the strategies could be assessed by prioritizing the inspection of screens potentially showing faults in addition to showing them, so that the identification of faults occurs faster. Besides, we believe the potential of action-effect strategies could be better exploited. For example, we could study the adoption of more fine-grained GUI screenshot distance for action-effect strategies instead of relying on traditional image distance. The distance between two GUI screenshots could be measured at the level of GUI layout and GUI component, which is adopted in a screen matching for test script repair study [66].

The proposed automated testing approaches are clearly beneficial since they automate testing in an average of 60.5% of the target instructions with a test budget of 1 hour, which is practical in the CI context. However, they do not enable testing all the instructions, which might still be the interest of developers. The problem could be solved by enhancing the components of the current approaches. To further improve ATUA effectiveness, part of our future work concerns the development of an additional set of reducers that will enable the ATUA's state abstraction function to distinguish between widgets containing different types of data. Another part of our future work is to integrate state-of-the-art solutions, particularly machine-learning-based techniques, to generate meaningful inputs unlocking "Gate States" [90] (i.e., states preventing the testing from going further without given some specific inputs) so that App exploration is more effective. One potential direction to improve ATUA is to study the impact of combining different solutions for dependency detection (including the dependencies between Windows and the dependencies between GUI components). It should be possible to deploy such solutions during static and dynamic analyses since, during the latter, ATUA could discover new relevant information.

Part of our future work is to leverage App Models inferred by testing to enhance static analysis. By analyzing the EWTG generated with static analysis and the EWTG derived after testing, we can identify mismatches between the models. This knowledge can be used later during the static analysis of the newer App version. A primary usage of it is to guide the static analysis tool associating GUI components with their handlers, which were missing due to the principal drawbacks of the static analysis.

Moreover, our probabilistic action sequences in CALM could be enhanced to prevent CALM from repeating a pattern in a context where it is not likely to succeed (e.g., when a setting is turned off, certain features would be hidden). To achieve this, one direction would be to deploy two probabilistic models at the same time, which is a common solution to avoid bias decisions (e.g., Double Q-Learning [127] deploys two Q-Tables to deal with the overestimation problems in Q-Learning).

Finally, inspired by an existing technique that translates generated test inputs into Espresso tests [128], part of our future work is to enable exporting test scripts (e.g., Espresso) as part of the output of our approaches. Engineers could improve test inputs by modifying the test scripts. Further, we may extend CALM to reuse existing test cases as test input sequences, which may enable CALM to explore App states difficult to reach otherwise (e.g., because specific domain knowledge is needed to select test inputs).

Bibliography

[1] Paolo Calciati, Konstantin Kuznetsov, Xue Bai, and Alessandra Gorla. What did really change with the new release of the app? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, pages 142–152, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Daniel Domínguez-Álvarez and Alessandra Gorla. Release practices for ios and android apps. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, WAMA 2019, pages 15–18, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Stuart Mcilroy, Nasir Ali, and Ahmed E. Hassan. Fresh apps: An empirical study of frequently-updated mobile apps in the google play store. *Empirical Softw. Engg.*, 21(3):1346–1370, June 2016.

[4] & Website (techbeacon.com) & HPE & Micro Focus Capgemini, & Sogeti. Proportion of budget allocated to quality assurance and testing as a percentage of it spend from 2012 to 2019, October 31, 2019. Last visited: 21/09/2013.

[5] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, Sept 2017.

[6] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66, 2019.

[7] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. QBE: QLearning-Based Exploration of Android Applications. IEEE, 4 2018.

[8] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of Android applications. ACM, 7 2020.

[9] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. IEEE, 11 2019.

[10] Chao Peng, Zhao Zhang, Zhengwei Lv, and Ping Yang. Mubot: Learning to test large-scale commercial android apps like a human. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 543–552, 2022.

[11] Faraz Yazdani Banafshe Daragh and Sam Malek. Deep gui: Black-box gui input generation with deep learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 905–916, 2021.

[12] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Deep Reinforcement Learning for Black-box Testing of Android Apps. *ACM Transactions on Software Engineering and Methodology*, 31, 7 2022.

[13] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 429–440, Washington, DC, USA, 2015. IEEE Computer Society.

[14] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. An empirical study of android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 738–748, New York, NY, USA, 2018. ACM.

[15] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. Detreduce: Minimizing android gui test suites for regression testing. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 445–455, New York, NY, USA, 2018. Association for Computing Machinery.

[16] Aman Sharma and Rupesh Nasre. Qadroid: Regression event selection for android applications. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 66–77, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Nana Chang, Linzhang Wang, Yu Pei, Subrota K. Mondal, and Xuandong Li. Change-Based Test Script Maintenance for Android Apps. IEEE, 7 2018.

[18] Cong Li, Yanyan Jiang, and Chang Xu. Cross-Device Record and Replay for Android Apps. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 395–407, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. UI Test Migration Across Mobile Platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 756–767, 2021.

[20] Z. Gao, C. Fang, and A. M. Memon. Pushing the limits on automation in gui regression testing. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 565–575, Nov 2015.

[21] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[22] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[23] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. Detecting and fixing data loss issues in android apps. In *Proceedings of the 31st ACM SIG-*

SOFT International Symposium on Software Testing and Analysis, ISSTA 2022, page 605–616, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. Data loss detector: automatically revealing data loss bugs in android apps. pages 141–152. ACM, 7 2020.

[25] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes: spotting ui display issues via visual understanding. pages 398–409. ACM, 12 2020.

[26] Mattia Fazzini and Alessandro Orso. Automated cross-platform inconsistency detection for mobile apps. pages 308–318. IEEE, 10 2017.

[27] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. Understanding and finding system setting-related defects in android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 204–215, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. Characterizing and finding system setting-related defects in android apps. *IEEE Transactions on Software Engineering*, 49:2941–2963, 4 2023.

[29] Camilo Escobar-Velásquez, Michael Osorio-Riaño, Juan Dominguez-Osorio, Maria Arevalo, and Mario Linares-Vásquez. An empirical study of i18n collateral changes and bugs in guis of android apps. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 581–592, 2020.

[30] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. Detecting non-crashing functional bugs in android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 434–446, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. An empirical study of functional bugs in android apps.

In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1319–1331, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal*, 27(1):149–201, 2019.

[33] Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. Automated, Cost-Effective, and Update-Driven App Testing. *ACM Trans. Softw. Eng. Methodol.*, 31(4), jul 2022.

[34] Wei Yang, Mukul R Prasad, and Tao Xie. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 250–265, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[35] Young-Min Baek and Doo-Hwan Bae. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 238–249, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 245–256, New York, NY, USA, 2017. ACM.

[37] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical gui testing of android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 269–280. IEEE Press, 2019.

[38] Domenico Amalfitano, Nicola Amatucci, Atif M. Memon, Porfirio Tramontana, and Anna Rita Fasolino. A general framework for comparing automatic testing techniques of Android mobile apps. *Journal of Systems and Software*, 125:322–343, 2017.

[39] Bill Phillips and Brian Hardy. *Android Programming: The Big Nerd Ranch Guide*. Big Nerd Ranch, 1st edition, 2013.

[40] Android.com. Intent Resolution. https://developer.android.com/reference/android/content/Intent, 2020. Last visited: 11/11/2020.

[41] Google for Developers. App manifest overview, 2023. https://developer.android.com/guide/topics/manifest/manifest-intro[Accessed: 14/11/2023].

[42] Android.com. Monkey - Android ui/application exerciser. http://developer.android.com/tools/help/monkey.html, 2020. Last visited: 12/17/2018.

[43] Nataniel P. Borges Jr., Jenny Hotzkow, and Andreas Zeller. Droidmate-2: A platform for android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 916–919, New York, NY, USA, 2018. ACM.

[44] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 469–480, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[46] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. Fastbot2: Reusable Automated Model-Based GUI Testing for Android Enhanced by Reinforcement Learning. In *37th IEEE/ACM International Conference on Automated Software Engineering*, ASE22, New York, NY, USA, 2023. Association for Computing Machinery.

[47] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of Android applications. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 153–164, 2020.

[48] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. QBE: QLearning-Based Exploration of Android Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 105–115. IEEE, apr 2018.

[49] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. Combodroid: Generating high-quality test inputs for android apps via use case combinations. *Proceedings - International Conference on Software Engineering*, pages 469–480, 2020.

[50] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 481–492, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Christophe Andrieu, Nando de Freitas, A. Doucet, and Michael I. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50:5–43, 2004.

[52] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[53] Robotium - github. https://github.com/RobotiumTech/robotium. Last visited: 12/10/2023.

[54] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[55] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, page 737–744, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[56] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th*

IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1070–1073, 2019.

[57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997.

[58] F. Song, Z. Xu, and F. Xu. An xpath-based approach to reusing test scripts for android applications. In *2017 14th Web Information Systems and Applications Conference (WISA)*, pages 143–148, 2017.

[59] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[60] Raja Vallée-Rai, Patrick Lam, Clark Verbrugge, Patrice Pominville, and Feng Qian. Soot (poster session): A java bytecode optimization and annotation framework. In *Addendum to the 2000 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)*, OOPSLA '00, pages 113–114, New York, NY, USA, 2000. ACM.

[61] Quan Chau Dong Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. Redroid: A regression test selection approach for android applications. In *Procedings of the 28th International Conference on Software Engineering and Knowledge Engineering*, SEKE 2016, pages 486–491, 2016.

[62] Quan Do, Guowei Yang, Meiru Che, Darren Hui, and Jefferson Ridgeway. Regression test selection for android applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 27–28, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[64] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Energy-aware test-suite minimization for android apps. *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 425–436, 2016.

[65] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li. Gui-guided repair of mobile test scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 326–327, 2019.

[66] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2020.

[67] X. Li, N. Chang, Y. Wang, H. Huang, Y. Pei, L. Wang, and X. Li. Atom: Automatic maintenance of gui test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171, 2017.

[68] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 55–66, New York, NY, USA, 2014. Association for Computing Machinery.

[69] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. Association for Computing Machinery.

[70] K. Rubinov and L. Baresi. What are we missing when testing our android apps? *Computer*, 51(4):60–68, April 2018.

[71] William M McKeeman. Differential testing for software. *Digital Technical Journal Vol.10 No.1 1998*, pages 100–107, 1998.

[72] Marco Mobilio, Oliviero Riganelli, Daniela Micucci, and Leonardo Mariani. Filo: Fix-locus localization for backward incompatibilities caused by android framework upgrades. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1292–1296, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Tsong Yueh Chen, Shing Chi Cheung, and Siu Ming Yiu. Metamorphic testing: A new approach for generating next test cases. Technical report, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

[74] Ke Chen, Yufei Li, Yingfeng Chen, Changjie Fan, Zhipeng Hu, and Wei Yang. Glib: Towards automated test oracle for graphically-rich applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1093–1104, New York, NY, USA, 2021. Association for Computing Machinery.

[75] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 143–152, New York, NY, USA, 1998. Association for Computing Machinery.

[76] S. Shamshiri, G. Fraser, P. Mcminn, and A. Orso. Search-based propagation of regression faults in automated regression testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 396–399, March 2013.

[77] Fabrizio Pastore, Leonardo Mariani, Antti E. J. Hyvärinen, Grigory Fedyukovich, Natasha Sharygina, Stephan Sehestedt, and Ali Muhammad. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 37–48, New York, NY, USA, 2014. Association for Computing Machinery.

[78] Minzhi Yan, Hailong Sun, and Xudong Liu. Itest: Testing software with mobile crowdsourcing. In *Proceedings of the 1st International Workshop on Crowd-Based Software Development Methods and Technologies*, CrowdSoft 2014, pages 19–24, New York, NY, USA, 2014. Association for Computing Machinery.

[79] STH blog editors. 10 Most Popular Crowdsourced Testing Companies in 2020. https://www.softwaretestinghelp.com/crowdsourced-testing-companies/. Last visited: 07/07/2020.

[80] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 342–351, 2013.

[81] Deloitte. 2018 Global Mobile Consumer Survey: US Edition. https://www2.deloitte.com/content/dam/Deloitte/us/Documents/technology-media-telecommunications/us-tmt-global-mobile-consumer-survey-exec-summary-2018.pdf.

[82] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for Android. *International Journal of Automated Software Engineering*, 25(4):833–873, December 2018.

[83] L. D. Toffola, C. Staicu, and M. Pradel. Saying 'hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–49, Oct 2017.

[84] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. Association for Computing Machinery.

[85] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 143–153, New York, NY, USA, 2014. Association for Computing Machinery.

[86] Android.com. Logcat command line tool. https://developer.android.com/studio/command-line/logcat, 2020.

[87] Fabrizio Pastore, Leonardo Mariani, Alberto Goffi, Manuel Oriol, and Michael Wahler. Dynamic analysis of upgrades in c/c++ software. In *Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering*, ISSRE '12, pages 91–100, USA, 2012. IEEE Computer Society.

[88] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 24th ACM Conference on Computer and Communication Security (CCS 2017)*. ACM, 2017.

[89] K. Kuznetsov, V. Avdiienko, A. Gorla, and A. Zeller. Analyzing the user interface of android apps. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 84–87, 2018.

[90] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. Combining automated gui exploration of android apps with capture and replay through machine learning. *Information and Software Technology*, 105:95–116, 2019.

[91] Chanh Duc Ngo, Fabrizio Pastore, and Lionel Briand. Atua replicability package. https://dropit.uni.lu/invitations/?share=e84048829ceddcb603ca, 2020. Last visited: 11/05/2021.

[92] Google Inc. Animator | android developers. https://developer.android.com/reference/kotlin/android/animation/Animator, 08 2020. Last visited: 11/11/2020.

[93] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[94] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 751–762, New York, NY, USA, 2016. Association for Computing Machinery.

[95] Jamendo. Music streaming app. https://www.jamendo.com/, 2020.

[96] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed.

In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[97] CNRS INRIA. Grid5000 infrastructure. https://www.grid5000.fr. Last visited: 07/07/2020.

[98] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 1, New York, New York, USA, 2011. ACM Press.

[99] Andrea Arcuri and Lionel Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[100] András Vargha and Harold D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[101] Tianxiao Gu. MiniTracing, APE coverage tool. http://gutianxiao.com/ape/install-mini-tracing. Last visited: 07/07/2020.

[102] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pages 560–564, 2015.

[103] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. An exploratory study of field failures. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 67–77, 2017.

[104] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - An initial survey. *SEKE 2010 - Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, pages 374–379, 2010.

[105] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*, volume 9783642290. 2012.

[106] Paul Ralph and Ewan Tempero. Construct Validity in Software Engineering Research and Software Metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, volume Part F1377, pages 13–23, New York, NY, USA, jun 2018. ACM.

[107] Sergio Di Martino, Anna Rita Fasolino, Luigi Libero Lucio Starace, and Porfirio Tramontana. Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing. *Software Testing Verification and Reliability*, pages 1–27, 2020.

[108] Gwowen Shieh, Show Li Jan, and Ronald H. Randles. On power and sample size determinations for the Wilcoxon-Mann-Whitney test. *Journal of Nonparametric Statistics*, 18(1):33–43, 2006.

[109] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering*, 42(2):170–186, 2016.

[110] Zebao Gao, Chunrong Fang, and Atif M. Memon. Pushing the limits on automation in gui regression testing. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 565–575, 2015.

[111] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-Travel Testing of Android Apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 481–492, New York, NY, USA, 2020. Association for Computing Machinery.

[112] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2020.

[113] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.

[114] Wunan Guo, Liwei Shen, Ting Su, Xin Peng, and Weiyang Xie. Improving Automated GUI Exploration of Android Apps via Static Dependency Analysis. pages 557–568. Institute of Electrical and Electronics Engineers Inc., 9 2020.

[115] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. Frauddroid: automated ad fraud detection for android apps. pages 257–268. ACM, 10 2018.

[116] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[117] R. N. Zaeem, M. R. Prasad, and S. Khurshid. Automated generation of oracles for testing user-interaction features of mobile apps. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 183–192, March 2014.

[118] D.G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, 1999.

[119] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[120] OpenCV. Histogram comparison. Last visited: 21/09/2023.

[121] R. Brunelli and O. Mich. Histograms analysis for image retrieval. *Pattern Recognition*, 34:1625–1637, 8 2001.

[122] Activity diary - github. https://github.com/ramack/ActivityDiary. Last visited: 12/10/2023.

[123] Amaze file manager - github. https://github.com/TeamAmaze/AmazeFileManager. Last visited: 12/10/2023.

[124] Ting Su, Jue Wang, and Zhendong Su. Benchmarking automated gui testing for android against real-world bugs. pages 119–130. ACM, 8 2021.

[125] Appium.io. Appium Documentation. http://appium.io, 2023. Last visited: 07/31/2023.

[126] Oliviero Riganelli, Marco Mobilio, Daniela Micucci, and Leonardo Mariani. A benchmark of data loss bugs for android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 582–586, 2019.

[127] Hado van Hasselt. Double q-learning. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, NIPS'10, page 2613–2621, Red Hook, NY, USA, 2010. Curran Associates Inc.

[128] Iván Arcuschin, Christian Ciccaroni, Juan Pablo Galeotti, and José Miguel Rojas. On the feasibility and challenges of synthesizing executable espresso tests. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, AST '22, page 92–102, New York, NY, USA, 2022. Association for Computing Machinery.